CECOM

CENTER FOR SOFTWARE ENGINEERING

ADVANCED SOFTWARE TECHNOLOGY

Subject: **Final Report - Real-Time Requirements Annex for Ada Reusability**

CIN: **C02 092LA 0010 00**

25 April 1990

REPORT DOCUMENTATION PAGE

Form Approved
OPM No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	25 April 1990	Final Report

4. TITLE AND SUBTITLE
Real-Time Requirements Annex for Ada Reusability

5. FUNDING NUMBERS
DAAB07-85-C- K524
TASK 7067-34040

6. AUTHOR(S)
Charles J. Albright, Mitchell J. Bassman, Carl Dahlke, Anthony Gargaro

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Computer Sciences Corporation
1 North Revmont Drive, Suite 30
Shrewsbury, NJ 07702

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
U. S. Army HQ CECOM
Center for Software Engineering
Fort Monmouth, NJ 07703-5000

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
STATEMENT A UNLIMITED

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

A set of guidelines for writing reusable software parts that satisfy some of the more common characteristics of real-time applications are provided. The emphasis is on coding aspects for developing reusable parts that include time dependency issues. It is structured as a stand-alone annex to a Reusability Handbook, originally developed by Computer Sciences Corporation (CSC) and enhanced by CECOM, that didn't consider any time dependency issues. The Annex contains four sections that provide a context for understanding the guidelines. The issues discussed in these sections are: distributed execution, task scheduling, error recovery, and resource control. There are five categories of guidelines included in the Annex: transportability, runtime dependency, composition orthogonality, readability, and efficiency.

14. SUBJECT TERMS	15. NUMBER OF PAGES
REAL-TIME, REUSE, TIME DEPENDENCY, ADA	88
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

Real-Time Requirements Annex

to the

Ada Reusability Handbook

prepared for

U.S. Army HQ CECOM
Center for Software Engineering
Advanced Software Technology
Fort Monmouth, NJ 07703-5000

prepared by

Charles J. Albright
Mitchell J. Bassman
Carl Dahlke
Anthony Gargaro

Computer Sciences Corporation
1 North Revmont Drive, Suite 30
Shrewsbury, NJ 07702

date

31 December 1989

Table of Contents

Annex A.....1

1. Distributed Execution.....3
   1.1. Partitioning.....4
   1.2. Virtual Nodes.....6
   1.3. Abstract Partitions.....9

2. Scheduling.....12
   2.1. Dynamic Task Priorities.....13
   2.2. Priority Queues.....18
   2.3. Predictable Scheduling.....24
   2.4. Rendezvous Optimizations.....27

3. Error Recovery.....31
   3.1. Reliable Transactions.....31
      3.1.1. Types of Reusable Real-Time Components.....33
         3.1.1.1. Actor and Server Components.....33
         3.1.1.2. Atomic and Composite Components.....34
      3.1.2. Composition of Components.....36
         3.1.2.1. Deadlock.....36
         3.1.2.2. Starvation.....44
         3.1.2.3. Thrashing.....46
   3.2. Fault Tolerance.....48
      3.2.1. Tolerance of Task Failure.....49
      3.2.2. Tolerance of Software Design Flaws.....52
         3.2.2.1. Recovery Blocks.....53
         3.2.2.2. N-Version Programming.....56
      3.2.3. Semantics of Failure in Distributed Ada Prog..57
   3.3. Asynchronous Transfer of Control.....60

4. Resource Control.....66
   4.1. Task Identification.....66
   4.2. Interrupts.....69
   4.3. Monitors and Semaphores.....72
      4.3.1. Binary Semaphores.....72
      4.3.2. General Semaphores.....76
      4.3.3. Monitors.....78
   4.4. Storage Reclamation and Reuse.....81

References.....86

Annex A

Real-Time Requirements

This annex to the CSC Ada Reusablility Handbook (ARH) [CSC87] provides a set of guidelines for writing reusable software parts that are required to satisfy some of the more common characteristics of real-time applications.

The annex contains four sections that address the issues of distributed execution, task scheduling, error recovery, and resource control. The four topical sections summarize the major issues pertinent to writing reusable parts for real-time requirements. These sections are not exhaustive in their treatment of real-time application characteristics, but provide a context for understanding the guidelines associated with each section.

The emphasis in these guidelines differs from that in the ARH. Whereas the guidelines in the ARH emphasized time-independent coding aspects of writing reusable parts, the annex guidelines emphasize coding aspects for developing reusable parts that include time dependency issues.

The ARH uses four categories of guidelines: transportability, runtime dependencies, composition orthogonality, and readability, which are also used in this annex. The following definitions are condensed from the ARH.

Transportability guidelines deal with criteria for writing components to make it possible to "... move a software part to a different execution environment, obtaining equivalent execution behavior, and minimizing the source modifications necessary to accomplish this. ... execution is equivalent in the sense that it complies with the rules of the RM. The execution, however, may not be identical, e.g., a series of values printed by the entries of a select statement may vary depending on the scheduling policy used in differing environments."

Runtime dependency guidelines deal with "the degree to which it is possible to reuse a software part in the same or different execution environment, obtaining identical execution behavior, and minimizing the source modifications necessary to accomplish this. Typical issues of concern include task scheduling policy, interrupt response time, and system initialization. Execution must be identical in the sense that there is no discernible difference in execution by any criteria of interest."

Composition orthogonality guidelines deal with " ... the degree to which a component is independent of the context in which it appears. For example, a subprogram that interfaces with its context through the use of global variables is much more dependent on its context than one that achieves the same interface through the use of parameters."

Readability guidelines deal with "the degree to which a part can be understood and subsequently maintained."

The issues addressed in this Real-Time Annex have led to the creation of another guideline category, the efficiency guideline. Efficiency guidelines address the concern for meeting timing deadlines in a hard real-time environment by indicating choices that can be made for designing real-time Ada components that take advantage of some understanding of the compilation system to increase the component's execution speed. One type of efficiency guideline deals with the use of known compiler optimizations to increase a component's efficiency. The benefits provided by engineering components to take advantage of such optimizations will be available when the component is reused in the original development environment. When the component is transported to another environment, the benefits will be available only if the other environment supports the same optimizations. The other type of efficiency guideline deals with alternatives that have consistent effects across compilation systems.

The need for a real-time programming discipline has been recognized for many years [ACM77] as essential if real-time programs are to be analytically verifiable and ultimately reliable. These attributes of verifiability and reliability are fundamental to reusable programs. Therefore, it is likely that reusability guidelines that aid in developing components meeting real-time requirements will provide a step towards formalizing a real-time programming discipline.

When execution time becomes an intrusive factor of the application, the domain of real-time programming is entered and further guidelines are warranted. The extent to which execution time intrudes into the function and performance of an application will often determine how much part reusability is attainable. When a part's execution time must satisfy a stringent upper bound to function correctly, the application is considered to have "hard" real-time requirements. Unfortunately, "hard " real-time requirements are frequently characteristic of applications that must execute in computers having limited processing resources. This characteristic further undermines part reuse, since language constructs that directly facilitate reuse may be crippled by a lack of processing capacity. For example, a numerically intensive algorithm may be most accurately formulated by using floating point types attributed to ensure optimal reuse. If, however, floating point processing is unavailable in the target computer, execution time may be unacceptable; an alternate formulation using fixed point types becomes necessary. Similarly, a complex data retrieval algorithm may be developed by using a type abstraction that depends upon access types to dynamically manage data storage. The reuse of the part may be restricted for real-time applications if storage reclamation costs are excessive due to space limitations.

In many instances, the time dependencies of "hard" real-time applications are less straightforward than meeting the execution time requirements of a serialized numerical computation. Typically, such applications include multiple cooperating programs that respond to many asynchronous and synchronous stimuli, where failure to respond is a potential threat to the continuing successful conduct of the application. Furthermore, in Mission Critical Computer Resource (MCCR) applications, the parts must be resilient to operational errors, computer failures, and deliberate hostile actions introduced to cause aberrant execution.

A reusable part for "hard" real-time applications must be rigorously documented according to a schema that formally describes the part's execution behavior. At a minimum, this behavior should be specified in terms of reusable attributes that identify the domain(s) of reuse [CEC88]. The reusable attributes, which are independent of the programming language, should address the critical time dependencies, performance efficiency, and resiliency of the part. In addition, language-dependent behavior attributes should be included to ensure a clear definition of the assumed capacity of the underlying processing resources, i.e., the Ada Virtual Machine. This level of documentation is in addition to that recommended in the ARH.

1. Distributed Execution

The requirement to distribute the execution of embedded real-time applications is becoming increasingly attractive with the availability of economical, powerful microprocessors and the rapid evolution of network communication technology. For MCCR systems, it is expected that many of the software engineering practices promoted by the Ada language will directly assist the distributed execution of embedded real-time application programs. This expectation is often substantiated by citing the note in the RM, refer 9 (5), that states parallel tasks may be implemented on multicomputers. Consequently, guidelines for writing reusable Ada parts should not preclude the potential distributed execution of an application that comprises reusable parts.

A comprehensive definition of a distributed execution environment depends upon such criteria as complexity, cost, function, and size. For this discussion, the following informal definition is derived from a restricted interpretation of the term "multicomputers" that characterizes the near-term environments emerging for embedded real-time MCCR applications. A distributed execution environment comprises multiple physically separate processing resources that may communicate and synchronize with each other without depending upon physically shared storage. The interpretation is further simplified by assuming that the processing resources conform to a consistent specification of the package SYSTEM. The literature often uses the term "node" to

designate a separate processing resource, as will the subsequent sections of the annex.

For the real-time application domain, a distributed execution environment offers several potential advantages over a centralized multiprocessor computer. The advantages correspond directly with many of the characteristics presented in the previously referenced definition. Major advantages include configurability, concurrency, and resiliency. Configurability facilitates adapting the application to the operational environment. For example, processing resources may be more easily connected to their data sinks/sources if they are not constrained to executing within a single computer. Concurrency facilitates improving application performance by achieving parallel execution. In many instances, concurrency may prevent the traditional scheduling problems that are often critical to the application domain. For example, periodic scheduling originates from the need to share a single processing resource to perform a specific functional allocation within precise time boundaries. The ability to perform each function on a separate processing resource transforms a difficult scheduling problem into a simpler problem of coordinating parallel functions. Resiliency facilitates protecting the application from computer failures. For example, degraded mode operation may often be instituted for an application when a processing resource fails in a distributed execution environment. This would not be an option if the processing resources were centralized in a single computer.

Each advantage partly results from viewing an application as a set of loosely coupled functions in which each function executes in its own separate customized execution environment. This view is in contrast to the more traditional view of a real-time application, in which the functionality is tightly coupled in a single execution environment with limited processing resources, such as avionics applications.

Unfortunately, these advantages are not automatically achieved by directly translating an application into a set of reusable Ada tasks, since an Ada task may not be a suitable unit on which to partition a given real-time application for a distributed execution environment [ACM87b, ACM88a]. Although partitioning by task has been tried with limited success [CEC89], in general it is necessary to identify an Ada program partitioning strategy that is best for the particular real-time application at hand. Ideally, such a strategy would facilitate combining reusable parts into a functional allocation that is compatible with a specific distributed execution environment.

1.1. Partitioning

Several approaches are proposed to partitioning Ada applications for distributed execution. To briefly review these approaches, the following paragraphs summarize four informal categorizations

of partitions: Main, Transparent, Subset , and Virtual Node Partitions.

Main Partitions - The unit for partitioning an application in this approach is an Ada main program, hence the name Main Partitions. This approach is a commonly used technique that requires an application to comprise multiple main programs and that each main program is dedicated to a separate node. In the near-term, this approach may predominate. It is independent of the Ada language, allowing an application to gradually transition from existing languages to Ada. Further, because communication among nodes is outside the Ada semantic model, the underlying nodal communication network may be directly used through user-defined packages. The principal disadvantage of this approach is the granularity for partitioning. The advantages of the Ada type model are compromised by the multiple program paradigm. Additionally, a rigid structure that is constrained by the distributed execution environment is forced upon the application. As a result, the application is less adaptable to nodal changes.

Transparent Partitions - The unit for partitioning an application in this approach is not restricted to specific Ada syntax or semantic boundaries. For example, a task need not execute on the same node as its master and declarative regions may extend across nodes. Therefore, a partition is transparent to the application. The principal advantage of this approach is that it realizes the full benefit of the Ada semantic model for distributed execution. However, it requires special tools to support the unrestricted partitioning. Further, implementations of the approach have shown that applications must assume a substantial overhead from the intrusion of additional constructs these tools generate. This overhead is unacceptable in the presence of real-time requirements.

Subset Partitions - The unit for partitioning an application is not restricted. However, only a subset of Ada may be used to avoid incurring the overhead of the Transparent Partitions. For example, the use of the terminate alternative might be precluded. While this approach has the same transparency advantage for the application as does that for Transparent Partitions, its utility depends upon limiting the use of Ada. Therefore, it has the disadvantage of promoting the proliferation of special Ada subsets.

Virtual Node Partitions - The unit for partitioning an application is the aggregation of cohesive Ada program units. Therefore, the application comprises a library of Ada program units that can be configured into partitions. This approach attempts to moderate the two extremes exemplified by Main Partitions and Transparent Partitions. Although both the application and partition retain the full benefit of the Ada semantic model of a single main program, the partitioning is not transparent to the application. The disadvantage of the approach is that an understanding of the nodal topology must necessarily constrain the de-

pendencies among partitions in order to reduce the communication processing overhead among remote nodes.

As a near-term practical alternative to Main Partitions, Virtual Node Partitions offer a partitioning approach that is adaptable to the stated rationale of the ARH. The properties that must be possessed by Virtual Nodes imply a methodology for structuring an Ada program that is conducive to distributed execution on one or more nodes. Consequently, guidelines for this methodology may be developed that are complementary to writing reusable parts.

In order to understand the relationship between Virtual Nodes and writing reusable parts, a synopsis of Virtual Nodes is presented in the following section.

1.2. Virtual Nodes

Virtual Nodes may be regarded as a language-independent application-oriented abstraction that permits expressing the disjoint distribution of the target execution environment in the composition of the application software. Since they represent the functional allocation to individual nodes, each Virtual Node encapsulates strongly cohesive units that are not reasonably executed on separate nodes. Consequently, Virtual Nodes are weakly coupled to each other so that communication overhead and synchronization overhead are minimized for real-time applications.

Four fundamental properties are associated with Virtual Nodes. These are:

1. Each Virtual Node must represent one (or more) distinct "thread of control(s)" that can be executed on a separate node;

2. Each Virtual Node must maintain its own internal state independent of the internal state of other Virtual Nodes;

3. Units shared among Virtual Nodes must have no internal state;

4. Communication interfaces among Virtual Nodes must not rely on access to shared variables.

In the adaptation of Virtual Nodes to Ada, the Ada library is viewed as comprising a collection of Virtual Nodes that may be configured for a distributed execution environment. This is significant since it emphasizes composing an application rather than partitioning an application. As a result, two independent steps become necessary to develop an application for a distributed execution environment: composing of an application using Virtual Nodes and configuring the Virtual Nodes. The first step concerns

the design and structure of Virtual Nodes in Ada. The second concerns transforming the resulting Virtual Nodes into executable components that can be distributed for execution and, in particular, the addition of remote communication support.

Obviously, writing reusable parts is complementary to achieving the first step. Further, the criterion for Composition Orthogonality is consistent with the properties of Virtual Nodes. Consequently, reusability guidelines must be formulated that specifically support the composition of Virtual Nodes. However, these guidelines must be subordinate to the conventions required to protect the fundamental properties of Virtual Nodes.

These conventions and their associated terminology are compatible with those adopted by the DIADEM project [CUP88], which has successfully applied them in partitioning "soft" real-time applications for distributed execution. These conventions permit viewing Virtual Nodes of arbitrary complexity as a collection of library units by restricting the sharing of units and the forms of communication among Virtual Nodes. In addition, these conventions specify a separate thread of control to permit independent execution and ensure that the library units are elaborated in a consistent order.

Unit Sharing - The sharing of objects among Virtual Nodes is precluded. Therefore, a library unit common to more than one Virtual Node must not specify visible object declarations that possess an internal state. Only type declarations, generic declarations, and subprogram declarations may be specified. Consequently, the units are termed template units, and they may include only other template units in their context clauses. Library units that do not conform to the above restrictions are termed nontemplate units, and they may be shared only within a Virtual Node.

Inter-Node Communication - The communication between Virtual Nodes is conducted through library units, or "interface packages." These units contain one or more tasks that define the remote entries provided to a Virtual Node. A Virtual Node may include as many interface packages as required. The assumption is that the Ada rendezvous is the preferred means for inter-node communication; this choice may not always be appropriate for "hard" real-time applications.

Independent Execution - The separate thread of control of a Virtual Node is achieved by specifying a procedure that acts as its root library unit. This unit, the Virtual Node root, includes a context clause that specifies all library units comprising the Virtual Node.

The following Ada code fragments provide a simplified illustration of the above conventions:

```
package Template_Package is
  ...
end Template_Package;

with Template_Package;
package Interface Package is

  task Communication_Task
    entry VN_1_Port (...);
  end Communication_Task;

end Interface_Package;

package VN_1_Package is
  ...
end VN_1_Package;

package VN_2_Package is
  ...
end VN_2_Package;

with VN_1_Package,
     Interface_Package,
     Template_Package;
procedure VN_1_Root is

begin
  ...
end VN_1_Root;

with VN_2_Package,
     Interface_Package,
     Template_Package;
procedure VN_2_Root is
begin
  Interface Package.Communication Task.VN_1_Port(...);
end VN_2_Root;
```

In the illustration, two procedures are defined as Virtual Node roots. Each procedure acts as the starting point of the Virtual Node's dependency graph specified by its context clause. The dependency graphs comprise a shared interface and template package, and a local nontemplate package. Consequently, the two Virtual Nodes may be distributed on separate nodes.

Guidelines for Virtual Nodes follow.

RTS Dependency Guideline 1.2-1: Avoid using remote conditional entry calls.

Discussion: The performance of a conditional entry call to a task encapsulated in an interface package depends upon the implementation of remote rendezvous. The term "immediately" in the defini-

tion of conditional entry calls is ambiguous in the context of distributed program execution. Therefore, the reuse of an Abstract Partition may be compromised by dependency upon the interpretation of a zero delay in the absence of a reliable abstraction for nodal time.

RTS Dependency Guideline 1.2-2: Avoid using remote timed entry calls.

Discussion: The performance of a timed entry call to a task encapsulated in an interface package depends upon the implementation of remote rendezvous. In the absence of a reliable abstraction of nodal time for distributed program execution, there is an ambiguity associated with the implementation's choice for starting the timed delay, i.e., when the call is issued or when the call is placed on the entry queue. Therefore, the reuse of an Abstract Partition may be compromised by a dependency on the precision of the timed delay.

RTS Dependency Guideline 1.2-3: Avoid using a zero delay in a timed entry call.

Discussion: The definition of a zero delay in a timed entry call requires starting the rendezvous immediately as for a conditional entry call. Where the part enclosing the entry call could be included in an Abstract Partition, there is a resulting difficulty in specifying part performance. This is a result of the method chosen by an implementation with respect to determining "immediacy" (refer Guideline 1.2-1). For example, a delay of 0.01 may fail, while a delay of 0.0 might succeed.

1.3. Abstract Partitions

Abstract Partitions are derived to support the Virtual Node approach. Although an Abstract Partition is notionally equivalent to a Virtual Node, in the absence of a unified concept for Virtual Nodes [ACM88a, AUK88], Abstract Partitions do not mandate a single form of inter-node communication. Any form of communication that is consistent with the criteria for software reuse may be employed. Consequently, if communication between different nodes is to be conducted using remote entry calls, then the resulting Virtual Nodes would not be identical to the Virtual Nodes that would result if remote procedure calls were used although derived from the same Abstract Partitions. The form of communication is dictated by the application domain, and reflected in the interface packages. Therefore, the principal role of Abstract Partitions is to restrict the sharing of remote data objects and to provide clearly defined remote communication interfaces. In this respect, that role may be perceived as complementary to the roles of Abstract Objects and Abstract Types, where the motivating rationale for Abstract Partitions is to control the latency and synchronization of remote access. Conse-

quently, construction of an Abstract Partition should emphasize combining reusable parts so that potential impediments to distributed execution are reduced or at least clearly specified.

Composition Orthogonality criteria provide the basis for formulating many specific guidelines for composing Abstract Partitions. Guidelines for Abstract Partitions follow.

Transportability Guideline 1.3-1: Avoid exchanging access objects among different Abstract Partitions.

Discussion: The value of access objects should not be made available as parameters to remotely visible entries since their representation is likely to be node dependent. Therefore, the reuse of an Abstract Partition is compromised when access object are used as inter-nodal parameters.

Composition Orthogonality Guideline 1.3-2: Avoid exchanging task objects among Abstract Partitions.

Discussion: The value of task objects should not be made available as parameters to remotely visible entries since doing so violates the composition rule that only tasks declared in interface packages may be visible among different Abstract Partitions.

RTS Dependency Guideline 1.3-3: Make available the use of a specific task entry within an interface package to ensure that an Abstract Partition does not terminate prematurely.

Discussion: Tasks declared in library units may terminate when the root procedure terminates, depending upon the implementation applicable to the node upon which the Abstract Partition resides. This would result in the premature termination of the Abstract Partition. Consequently, to increase the reusability of an Abstract Partition, a specific task entry should be specified within an interface task to allow for its orderly termination.

Transportability Guideline 1.3-4: Achieve synchronization among Abstract Partitions through task rendezvous.

Discussion: Entry calls and accept statements in tasks declared in interface packages should be used to synchronize the execution of Abstract Partitions. When Abstract Partitions reside on different nodes, real-time performance constraints may require a relaxation of this guideline.

RTS Dependency Guideline 1.3-5: Avoid using task attributes to reference a task in a different Abstract Partition.

Discussion: Using the 'Callable and 'Terminated attributes that reference a task in another Abstract Partition, i.e., an internode communication task, may yield unreliable results when the Abstract Partitions reside on different nodes.

Composition Orthogonality Guideline 1.3-6: Use a collection of library units to achieve distributed execution.

Discussion: A collection of library units that conform to a set of composition rules forms the partitioning unit for distributed execution. This partitioning unit, an Abstract Partition, allows the reuse of library units for different applications.

Composition Orthogonality Guideline 1.3-7: Ensure that the library units reused among Abstract Partitions do not contain or provide access to an object.

Discussion: Reusing library units among Abstract Partitions that compose a distributed program is permitted only if the units do not contain or provide access to the declaration of an object, where an object may be a variable, a task, a file, or a device. Consequently, because the units may not possess an internal state, they are independent of node residency. These units are termed template units.

Composition Orthogonality Guideline 1.3-8: Accomplish communication among Abstract Partitions through the specification of interface packages.

Discussion: Reusing Abstract Partitions requires that communication be performed only through entry and procedure calls. The declaration of the respective tasks and procedures must be encapsulated in a library unit termed an interface package. An Abstract Partition may include one or more interface packages.

Composition Orthogonality Guideline 1.3-9: Distribute a program through a root procedure that includes in its context clause all library units composing an Abstract Partition.

Discussion: The specification of an Abstract Partition requires the definition of a procedure that includes all library units composing the Abstract Partition in its context (with) clause. This procedure, the root procedure, is the start of the dependency graph for the Abstract Partition.

Composition Orthogonality Guideline 1.3-10: Ensure that template units do not include nontemplate units in a context clause.

Discussion: Any context (with) clauses of a template unit may include references only to other template units. This ensures that a template unit cannot have any side effects that compromise the stateless property of a template unit.

Composition Orthogonality Guideline 1.3-11: Ensure that nontemplate units include only library units that are included in the same Abstract Partition.

Discussion: There is an unresolved issue with respect to referencing communication tasks contained in interface packages by nontemplate units. How can a nontemplate unit have visibility to the entries of a different Abstract Partition?

Composition Orthogonality Guideline 1.3-12: Minimize complex elaboration dependencies among Abstract Partitions.

Discussion: The elaboration of Abstract Partitions may proceed independently on different nodes. Therefore, a minimum of dependencies should be made upon the order of this elaboration.

Composition Orthogonality Guideline 1.3-13: Raise an appropriately defined exception in the client task when an error condition is detected within the body of an accept statement for which no local recovery is possible.

Discussion: When a server task detects an error condition within the body of an accept statement for which no local recovery is possible, an exception indicative of the condition should be raised in the client task. Doing so is preferable to the alternative of returning notification of the error condition through an actual parameter of the accept statement. The latter strategy may unnecessarily delay recovery action by the client task while the server task is reaching the end of the rendezvous, particularly when the tasks reside in different abstract partitions. Furthermore, the use of an exception provides the opportunity for the server task to using an identical manifestation of the condition, i.e., the exception name.

2. Scheduling

The scheduling control of shared processing resources remains a principal requirement of many embedded real-time application programs. This requirement results from the need to transform complex timing constraints into simpler, more manageable processor utilization constraints. Until distributed execution environments mature and make shared processing obsolete, this requirement may limit the reuse of parts enclosing tasks. This limitation results because in many instances the normal Ada scheduling semantics do not adequately control the necessary transformations. Therefore, to partially overcome some of the difficulties task scheduling introduces in developing reusable parts, several issues are presented. From these issues, general guidelines and paradigms are derived that may eventually promote increased reuse of parts that depend upon some degree of control of task scheduling.

The particular issues are not exhaustive. Their discussion is motivated by ongoing initiatives to introduce support of formal real-time task scheduling algorithms [ACM87b, ACM88a] into the Ada language for tasks executing on a single processor computer.

In addition, the normal stipulation on avoiding RTS dependencies is relaxed, where necessary, in favor of explicitly specified dependencies. This is justified only when a dependency is essential to demonstrating that a part can be executed correctly. In other words, without the dependency, the part's execution would be nondeterministic.

2.1. Dynamic Task Priorities

A frequently cited requirement of "hard" real-time applications is that the application be able to control dynamically the execution priority of a thread of control. This capability would enable a task to change its own priority or that of another task, depending upon conditions encountered during execution. This explicit change of priority is separate from the requirement that the RTS change task priority when it is necessary to support predictable task scheduling algorithms, such as rate-monotonic, that depend upon a priority inheritance scheme for task execution in order to avoid the condition of "priority inversion" in an application.

When a reusable part is being written, any dependencies upon the priority of its enclosing task execution must be carefully specified, because the notion of priority may be supported quite differently in those execution environments in which the part may be reused. n addition, several issues are raised with respect to developing reusable Ada parts and dynamic task priorities. Two of the principal issues are antithetical, since they address introducing dynamic priorities into an application and avoiding assumptions about dynamic priorities.

The RM provides only limited support for specifying the execution priority of a task; task priority is static, and therefore fixed. The priority of a task cannot be changed, unless the task is engaged in a rendezvous with a higher priority task. The task's priority should be used only to convey the task's relative urgency in order for the RTS to assign processing resources. In addition, the specification of priority must be a static expression, thereby preventing its representation as a formal generic parameter. Consequently, applications that cannot operate under the constraints of fixed task priorities must apply reusable techniques to afford some control over task execution priority. One such technique that is consistent with the RM can be implemented as a reusable paradigm for a restricted form of dynamic task priorities. This technique exploits the use of the rendezvous for changing priority. Any task that is required to change its priority must conform to a set of rigid conventions for structuring the parts that are to execute at a specific priority by enclosing the parts in the body of an accept statement. Immediately prior to the accept statement, an entry call is made to

a "task priority manager" that activates a task at the required priority. The activated task includes an entry call to this accept.

The following reusable parts for dynamically controlling task priority are presented to detail some of the issues raised by the paradigm. These parts have been used to demonstrate that with appropriate synchronization control, task priorities can be successfully changed, resulting in the reassignment of processing resources in an interleaved task execution environment on a single processor. To reduce the Ada text, the example included only the essential framework. Similar text that can be derived from the immediately preceding text or the enclosing context is denoted by an ellipsis, and the use clause is included to abbreviate references that are readily apparent.

```
with SYSTEM;

package Priorities Data Package is

  Hi_Priority  : constant SYSTEM.Priority
               := SYSTEM.Priority'Last-1;
  Med_Priority : ... ;
  Low_Priority : ... ;
  type Task_Priority_Type  is (Low, Med, Hi);
  type Priority_Array_Type is array (Task_Priority_Type)
                                    of SYSTEM.Priority;
  Task Priorities : Priority_Array_Type
                  := (Hi  =>  Hi_Priority,
                      Med =>  Med_Priority,
                      Low =>  Low_Priority);

end Priorities_Data_Package;

with Priorities_Data_Package;
use  Priorities_Data_Package;
package User_Tasks_Package is

  task type User_Task_Type is
       entry Run (This_Task_Priority : in Task_Priority_Type);
       pragma Priority (Low_Priority);
  end User Task Type;

end User_Tasks_Package;



with Priorities_Data_Package;
use  Priorities_Data_Package;
with SYSTEM;
package Dynamic_Priorities_Package is

  type Task_Id_Type is limited private;
```

```
  task Priority_Task is
       entry Change_Priority
             (User_Task Id  : in Task_Id_Type;
              Task_Priority : in Task_Priority_Type);
       pragma Priority (SYSTEM.Priority'Last);
  end Priority_Task;

private
  -- Reuse:
  -- The following declarations ensure that a task
  -- cannot directly access the identity of another
  -- task.
  Global_Task_Id : Integer;
  Null_Task_Id   : Integer := 0;

  type Task_Id_Type is
    record
      Task_Id : Integer := Global_Task_Id;
    end record;

end Dynamic Priorities Package;


with User_Tasks_Package;
use  User_Tasks_Package;
-- Reuse:
-- Elaboration order must be explicitly specified since
-- elaboration of Dynamic Priorities Package is dependent
-- upon the elaboration of the body of User Task Type.
pragma Elaborate (User_Tasks_Package);

package body Dynamic_Priorities_Package is

  Task_Set_Size : constant := ...;
  subtype Task_Set_Subtype is Integer
                           range 1..Task_Set_Size;
  type User_Task_Type_Pointer is access User_Task_Type;
  type Task_Array_Type is array (Task_Set_Subtype)
                       of User_Task_Type_Pointer;

  User Tasks: Task_Array_Type;


  -- The following task types make an entry call to the
  -- reusable parts that are to execute at a specified
  -- priority. Objects of these task types are activated
  -- by the "task priority manager" after the reusable
  -- part has requested a change in priority.

  task type Hi_Task_Type is
       entry Get_Task  (Task_Id : in Task_Set_subtype);
       pragma Priority (Hi Priority);
```

```
end Hi_Task_Type;

task type Med_Task_Type is
     entry Get_Task  (Task_Id : in Task_Set_subtype);
     pragma Priority (Med_Priority);
end Med_Task_Type;

task type Low_Task_Type is
     entry Get_Task  (Task_Id : in Task_Set_subtype);
     pragma Priority (Low_Priority);
end Low_Task_Type;

type Hi_Task_Pointer_Type  is access Hi_Task_Type;
type Med_Task_Pointer_Type is access Med_Task_Type;
type Low_Task_Pointer_Type is access Low_Task_Type;


task body Hi_Task_Type is
  Client : Task_Set_Subtype;
begin
  accept Get_Task (Task_Id : in Task_Set_Subtype) do
    Client := Task_Id;
  end Get_Task;
  User_Tasks(Client).Run(Hi);
end Hi_Task_Type;


task body Med_Task_Type is ... ;
task body Low_Task_Type is ... ;
task body Priority_Task is
  Hi_Runner  : Hi_Task_Pointer_Type;
  Med_Runner : Med_Task_Pointer_Type;
  Low_Runner : Low_Task_Pointer_Type;
begin
  -- Reuse:
  -- The following task identification technique is
  -- reusable and is discussed later in the annex.
  for Task_Id in User_Tasks'Range loop
    Global_Task_Id      := Task_Id;
    User_Tasks(Task_Id) := new User_Task_Type;
  end loop;
  -- Real_Time Assertion:
  -- The priority of this task is higher than any
  -- task activated in the preceding loop. Therefore,
  -- it can be assumed, for logically parallel tasks,
  -- that the following assignment will be completed
  -- before the activated tasks commence execution.

  Global_Task_Id := Null_Task_Id;
  loop
    begin
      select
        accept Change Priority
```

```
                    (User_Task_Id   : in Task_Set_Subtype;
                     Task_Priority : in Task_Priority_Type) do
                Task_Set_Id := User_Task_Id.Task_Id;
                case Task_Priority is
                  when Hi =>
                       Hi_Runner := new Hi_Task_Type;
                       Hi_Runner.Get_Task (Task_Set_Id);
                  when Med =>  ... ;
                  when Low =>  ... ;
                end case;
              end Change Priority;
            or
              terminate;
            end select;
          exception
            when others =>
              raise;
          end;
        end loop;
      end Priority_Task;
    end Dynamic_Priorities_Package;

    with Dynamic_Priorities_Package;
    use  Dynamic_Priorities_Package;
    package body User_Tasks_Package is
      task body User_Task_Type is
        This_Task_Id : Task_Id_Type;
        This_Task_Priority : Task_Priority_Type := ...;

      begin
        Priority_Task.Change_Priority
                    (This_Task_Id, This_Task_Priority);
        accept Run (This_Task_Priority) do
        -- Part that is to be executed at dynamic priority.
        end Run;
      end User_Task_Type;
    end User_Tasks_Package;
```

It is important to understand the limitations of the above technique before reusing it. There is an inherent deficiency with respect to the immediacy of a task executing at a higher priority. Once the request to change priority has been accepted, an indeterminate delay may occur before the task proceeds to execute at the new priority, if other tasks are ready for execution with a priority greater than the static fixed priority of the task that has requested the change. There are several means of resolving this problem, provided that all tasks comply with an established convention that requires each task to voluntarily allow the priority change to become effective within some reasonable upper bound. Alternatively, all tasks in an application may be required to synchronize changes in task priority so that the highest priority change is serviced first. This would necessitate increased complexity in the "task priority manager."

This complexity may be avoided if all tasks, after having synchronized the changes in priority, include a delay as the first statement of the accept body so that the normal priority scheduling of task execution is applied.

Guidelines for Dynamic Task Priorities follow.

Transportability Guideline 2.1-1: Avoid assumptions regarding the default priority of task types.

Discussion: The default priority of task objects of the same task type need not be identical in the absence of the Priority pragma. Therefore, when tasks of the same type must be assigned identical priorities, the Priority pragma should be specified in the task type declaration.

Transportability Guideline 2.1-2: Avoid assumptions about the value of the default priority of a task type.

Discussion: The value of the default priority of a task type may differ among implementations. Therefore, when a task must be assigned a specific priority, such as the lowest priority, the Priority pragma should be used in the task type declaration.

Transportability Guideline 2.1-3: Avoid depending upon task priority to arbitrate among open accept alternatives of a selective wait statement.

Discussion: Multiple open alternatives of a selective wait statement are not arbitrated by the priority of tasks in the respective entry queues. While an implementation may choose to use task priority to select an accept statement, this is not guaranteed for all implementations. Therefore, when a reusable part requires some degree of control in arbitrating among multiple open accept statements, the part must explicitly define this arbitration.

2.2. Priority Queues

The development of real-time applications may be hindered by the requirement in the RM to perform queued task entry calls using a First-In-First-Out (FIFO) order. For example, if tasks are to be executed according to the Rate Monotonic Scheduling algorithm, the upper bound for the worst-case deadline for task execution is reduced if queued task entry calls are serviced according to task priority. Consequently, when writing a reusable part that services entry calls, it may be necessary to override the FIFO order. This must be accomplished without depending upon any special provisions provided through implementation defined pragmas or calls to the RTS.

A recommended approach to writing a part to service entry calls in a non-FIFO order is using entry families [RAT86]. This approach offers many variations to achieve different orders for servicing entry queues. Two such variants illustrate reusable parts that service entry queues based upon a task priority specified by the calling tasks. Both parts are refinements to that variant identified to support the Rate Monotonic Scheduling algorithm [ACM87b]. However, the additional rendezvous inherent in each part may reduce their utility for "hard" real-time requirements unless they are recognized by compilers as idioms that are open to extensive optimization.

The technique used by the first variant requires an additional task to ensure maintaining the prescribed order in the prioritized entry queue. Both the client and server tasks transparently rendezvous with this task. This is accomplished by encapsulating the queueing task in a generic package that provides the client and server with a procedural interface that is instantiated for each prioritized entry. In this regard, the technique is restrictive: the client task cannot use a selective wait statement in conjunction with a prioritized entry call. The client procedure, Request Service, registers a client task's requested priority registered for subsequent service by an entry family and enqueues it in FIFO order on the entry family queue associated with the requested priority. The client task is then blocked until the entry family accept statement to dequeue the client task is executed. That allows the server's accept statement to be called as a procedure. The priority queue task manages the registration of the client task priorities so that the entry family accepts are performed in order of priority. The server procedure causes the server task to be synchronized by the priority queue task so that it is always waiting to rendezvous with a client task. Otherwise, the priority order would be compromised and the potential for FIFO queueing would again prevail.

The technique used in the second variant is essentially the same as that of the first, except that an optimization has been applied. This optimization eliminates the server task by fusing the accept body of its prioritized entry into the priority queue task. Consequently, the server procedural interface is no longer required since the client task to be serviced is always properly synchronized because the dequeue request was accepted. The elimination of the server task introduces restrictions that may reduce the reusability of the part.

The two example variants follow.

```
-- Priority_Queue (Variant 1). This variant requires three
-- additional rendezvous in providing a reusable part for
-- prioritizing an entry queue.
-- Reuse:
-- The package must be instantiated for every entry queue
-- that is to be serviced in a prioritized order.

generic

  type Priority_Type is ( );
  type Entry_Params_Type is private;
  with procedure Entry_Procedure
                 (Entry_Params : in out Entry_Params_Type);

package Priority_Queue_Template is

  -- Reuse:
  -- The procedural interface for client tasks.

  procedure Request Service
            (Request_Priority : in Priority_Type;
            (Request_Params   : in out Entry_Params_Type);


  -- Reuse:
  -- The procedural interface for server tasks.
  -- A call to this procedure must precede the
  -- accept body for the prioritized entry.

  procedure Accept_Request;

end Priority_Queue_Template;


package body Priority_Queue_Template is

  task Priority_Queue_Task is
    entry Unblock_Server;
    entry Enqueue_Client
          (Request_Priority : in Priority_Type);
    entry Dequeue_Client (Priority_Type);
  end Priority_Queue_Task;

  task body Priority_Queue_Task is
    Next_Priority   : Priority_Type
                    := Priority_Type'First;
    Blocked_Clients : array (Priority_Type)
                      of Natural := (others => 0);
  begin
    loop
      select
        accept Enqueue_Client
          (Request_Priority : in Priority_Type) do
```

```
            Blocked_Clients(Request_Priority) :=
              Blocked_Clients(Request_Priority) + 1;
            if Request_Priority   Next_Priority then
               Next_Priority := Request_Priority;
            end if;
          end Enqueue_Client;
        else
          if Next_Priority = Priority_Type'First and
             Blocked_Clients(Priority_Type'First) = 0 then
            select
              accept Enqueue_Client
                      (Request_Priority: in Priority_Type) do
                Next_Priority := Request_Priority;
                Blocked_Clients(Request_Priority) := 1;
              end Enqueue_Client;
            or
              terminate;
            end select;
          end if;
          select
            accept Unblock_Server;

            -- Assertion:
            -- There is at least one client task waiting or is
            -- ready to wait for this entry.

            accept Dequeue_Client(Next_Priority);
            Blocked_Clients(Next_Priority) :=
                Blocked_Clients(Next_Priority) - 1;


            if Next_Priority > Priority_Type'First and
               Blocked_Clients(Next_Priority) = 0 then
               Next_Priority :=
                   Priority_Type'Pred(Next_Priority);
            end if;
          else null;
          end select;
        end select;
      end loop;
    end Priority_Queue_Task;

    procedure Request_Service
              (Request_Priority : in Priority_Type);
              (Request_Params   : in out Entry_Params_Type) is
    begin

      Priority_Queue_Task.Enqueue_Client (Request_Priority);
      Priority_Queue_Task.Dequeue_Client (Request_Priority);
      -- Assertion:
      -- There can be no task already waiting on the
      -- entry bound to the procedure.
      Entry Procedure (Request_Params);
```

```
    end Request_Service;

    procedure Accept_Request is
    begin

      Priority_Queue_Task.Unblock_Server;

    end Accept_Request;

  end Priority_Queue_Template;



  -- Priority_Queue (Variant 2). This variant requires one
  -- additional rendezvous in providing a reusable part for
  -- prioritizing an entry queue.
  -- Reuse:
  -- The package must be instantiated to replace each entry
  -- that is to be serviced in a prioritized order. The
  -- accept body of the replaced entry becomes a service
  -- procedure that is bound to an entry family.
  generic

    type Priority_Type is ( );
    type Entry_Params_Type is private;
    with procedure Entry_Procedure
                   (Entry_Params : in out Entry_Params_Type);
  package Priority_Queue_Template is
    -- Reuse:
    -- The procedural interface for client tasks.

    procedure Request_Service
              (Request_Priority : in Priority_Type;
               Request_Params   : in out Entry_Params_Type);

  end Priority_Queue_Template;

  package body Priority_Queue_Template is
    task Priority_Queue_Task is
      entry Unblock_Server;
      entry Enqueue_Client
            (Request_Priority : in Priority_Type);
      entry Dequeue_Client
            (Request_Priority : in Priority_Type;
            (Request_Params   : in out Entry_Params_Type);

    end Priority_Queue_Task;

    task body Priority_Queue_Task is
      Next_Priority   : Priority_Type
                        := Priority_Type'First;
      Blocked_Clients : array (Priority_Type)
```

```
                        of Natural := (others => 0);
    begin
      loop
        select
          accept Enqueue_Client
            (Request_Priority : in Priority_Type) do
            Blocked_Clients(Request_Priority) :=
            Blocked_Clients(Request_Priority) + 1;
            if Request_Priority > Next_Priority then
               Next_Priority := Request_Priority;
            end if;
          end Enqueue_Client;
        else
          if Next_Priority = Priority_Type'First and
             Blocked_Clients(Priority_Type'First) = 0 then
            select
              accept Enqueue_Client
                 (Request_Priority : in Priority_Type) do
                 Next_Priority := Request_Priority;
                 Blocked_Clients(Request_Priority) := 1;
              end Enqueue_Client;
            or
              terminate;
            end select;
          end if;
          -- Assertion:
          -- There is at least one client task waiting or is
          -- ready to wait for this entry.

          accept Dequeue_Client(Next_Priority)
            (Request_Params : in out Entry_Params_Type) do
            -- Call service to be performed for client task.
            Entry_Procedure (Request_Params);
          end Dequeue_Client;
          Blocked_Clients(Next_Priority) :=
            Blocked_Clients(Next_Priority) - 1;
          if Next_Priority > Priority_Type'First and
             Blocked_Clients(Next_Priority) = 0 then
            Next_Priority := Priority_Type'Pred(Next_Priority);
          end if;
        end select;
      end loop;
    end Priority_Queue_Task;

    procedure Request_Service
              (Request_Priority : in Priority_Type;
               Request_Params   : in out Entry_Params_Type) is

    begin

      Priority_Queue_Task.Enqueue_Client (Request_Priority);
      Priority_Queue_Task.Dequeue_Client
              (Request_Priority, Request_Params);
```

```
    end Request_Service;

  end Priority_Queue_Template;
```

2.3. Predictable Scheduling

An important characteristic of embedded real-time applications is the ability to predict reliably the scheduling of task execution. This predictability is at risk when an application is composed from reusable parts that contain tasks. The correct scheduling of a task in one application is no guarantee that it will be correctly scheduled in a new application unless the parts are designed to enforce a task scheduling discipline that can be proved predictable. As a result, composing deadline-driven or "hard" real-time applications from reusable parts requires using formal scheduling theory guidelines, particularly when the application is targeted for a different Ada RTS from that used to develop the parts originally.

While, in the near-term, specific guidelines cannot be formulated to guarantee the predictability of task scheduling, emerging techniques derived from priority inheritance protocols provide the basis for guidelines that may have future application. One such technique is the priority ceiling protocol. This protocol facilitates the reuse of parts for a limited, but useful, domain of real-time applications by carefully separating timing issues from predefined logical paradigms for tasks composing an application. However, since this protocol has been proven only under controlled conditions, the claimed predictability and improved processor utilization should be carefully reviewed for each application.

The premise for priority inheritance protocols is to reduce the occurrence of priority inversion in an Ada application that uses tasks of different priorities. Priority inversion arises when a low priority task can indefinitely block the execution of a higher priority task. The priority ceiling protocol minimizes this blocking time within predictable bounds and prevents non-trivial forms of tasking deadlock. The overall effect is that higher priority tasks complete earlier than under the basic priority inheritance protocol. Applying the protocol, requires the following rules on the use of Ada tasks:

1. Each non-server task must be assigned a priority.
2. Server tasks must not be assigned a priority.
3. Conditional and timed entry calls may not be used.
4. Server tasks comprise a single continuous loop that encloses an unguarded select statement.
5. The select statement may enclose only accept statements.
6. Nested accept statements may not be used.

The above rules are stated in terms of client tasks and server tasks. A server is in essence a semaphore that controls entries to critical regions, and a client is simply a task that calls these entries. Further, only server tasks may enclose accept statements, and a client task must enclose at least one entry call to a server task. A non-server task is a task that does not enclose accept statements. Consequently, the set of client tasks is not equivalent to the set of non-server tasks. A typical server task has the following structure:

```
task type Server_Task_Type is
  entry Critical_Region_1 (...);
  ...
  entry Critical_Region_n (...);
end Server Task Type;



task body Server_Task_Type is
begin
  loop
    select
      accept Critical_Region_1 (...) do
        ...
      end Critical_Region_1;
    ...
    or
      accept Critical_Region_n (...) do
        ...
      end Critical_Region_n;
    end select;
  end loop;
end Server_Task_Type;
```

The protocol ensures that a task's execution can be delayed only by its pre-emption by a higher priority non-server task (or the execution of a server on behalf of such a client task), or by its being blocked by a server executing on behalf of a lower priority task. This blocking is limited to, at most, the longest call made by a lower priority client. In this context a server is defined as "executing on behalf of" a client task if either the client is on an entry queue of, or is in rendezvous with, the server, or if the server has been called by a server executing on behalf of the client task.

The protocol derives its name from associating each server task with the highest (ceiling) priority of any client task that may call it. This ceiling priority is used by the RTS to prioritize entry calls. In particular, it is used to determine whether or not to block the execution of the calling client task. This use of priority ceiling and the changing of the priority of server tasks places specific requirements (dependencies) upon the RTS

implementation. The dependencies affect the atomicity of processing entry calls, the adoption of client priorities, and the inheritance of non-server priorities by server tasks.

Three types of blocking may occur using the protocol. They are defined as:

1. Direct blocking: the server task is executing on behalf of a client task; the client is blocked according to normal Ada rules.

2. Push-through blocking: a server task is executing on behalf of a lower priority client task with an inherited priority of a high priority client task; this situation may delay the execution of a medium priority task.

3. Ceiling blocking: a server task is executing with a priority ceiling that is greater than or equal to that of the client task; the client task is blocked.

The rules for constructing Ada tasks and the above definitions may be used to describe the priority ceiling protocol informally. The description is presented with respect to the actions associated with the occurrence of a pre-emptive condition, reaching an entry call, reaching a select statement, and reaching the end of an accept statement. In addition, tasks are assumed to be in one of three states: blocked, executing, or schedulable.

1. When a pre-emptive condition occurs, such as the expiration of a delay statement by a higher priority non-server task, the currently executing task becomes schedulable and the highest priority schedulable task is then executed.

2. When a select statement is reached and no client task is currently queued for service, the server task is blocked according to normal Ada rules and the highest priority non-blocked task is scheduled for execution. Once a non-server task is schedulable, a select statement cannot be reached without a call having been made by a client task.

3. When an entry call is made by a client task, the client task is blocked. The type of blocking is then defined depending upon the state of all server tasks. If no server task is executing on behalf of any client task, the server task adopts the client task priority and becomes schedulable for execution. This situation results in direct blocking of the client task. If one or more server tasks are executing on behalf of client tasks, the priority ceiling of each server task is compared to that of the immediate calling client task. If

the client priority is greater, the conditions are equivalent to no server task executing on behalf of a client task. Otherwise, the priority of the highest priority blocked client task is inherited by the server executing on behalf of the highest priority client, and the server task becomes schedulable for execution. This situation results in priority ceiling blocking of the client task, and may cause push-through blocking if a medium priority task becomes blocked.

4. When the end of an accept statement is reached, the highest priority task blocked by the server task enclosing the accept statement is unblocked and becomes schedulable for execution.

One rule for constructing server tasks is that they must not be assigned an explicit priority. This rule permits the server tasks to be scheduled for execution, by interpreting RM 9.8 (5) with respect to the statements "the scheduling rules are not defined" and "at least that priority," so that a server task may reliably pre-empt a client task before the entry call is completed. It also enables a client task to conveniently inherit priorities.

The protocol ensures that only one client can be queued for a server task, that only self-imposed deadlock can occur, and that client tasks are unblocked in order of priority.

2.4. Rendezvous Optimizations

The time required to perform a rendezvous depends on several factors. In fact, the discussion of rendezvous optimization should probably address two topics: the rendezvous itself and the function served through the use of the rendezvous. The first topic is largely a language issue, initially the concern mainly of language designers and implementers of compilers and runtime environments. The application builder, of course, must understand the resulting rendezvous definitions and implementation to use them effectively. This leads to the second topic, which comprises application design strategies and Ada coding techniques. In a real-time application, the goal is to employ rendezvous to advantage safely and efficiently.

The basic Ada rendezvous is a synchronous, asymmetrical interaction of a calling (producer or client) task or subprogram with a called (consumer or server) task. It is synchronous because it can occur only when both parties are available at the same time. It is asymmetrical in that (1) a caller must know the name and specification of the called task, while the called task does not know the name of the caller(s); and (2) a caller may start only one rendezvous at a time while the called task may have a number of callers queued waiting for service.

In a typical implementation, an entry call causes the suspension of the client task and a subsequent context switch to the server task. When the server task completes execution of its accept statement, the client task is placed in a ready-to-execute state that is eventually followed by a context switch to the client task. Thus, at least two context switches are required to complete the rendezvous. Unfortunately, a context switch includes overhead processing that degrades system performance.

The optimization proposed by Habermann and Nassi [HAB80] addresses the context switch overhead problem. Briefly stated, this proposal would have the compiler recognize that certain server task entries could be replaced by a procedure, with the necessary calling synchronizations being performed through the use of low-level operating system event flags. It has been reported [BUR87] that, in one implementation, a procedure call is 45 times faster than a two-context-switch rendezvous, which makes this approach to optimization look very attractive.

The general Habermann and Nassi algorithm handles cases that include select statements with delay and else alternatives, nested accept statements, and exceptions raised during rendezvous. This algorithm reduces but does not eliminate all context switching. However, in some simpler situations it may be possible to eliminate the server task completely if the rendezvous code is structured appropriately. One of the requirements is that all server task code executed as a result of a rendezvous be positioned within the accept statement. Therefore, all other code in the task can be considered, by default, to be administrative code that is of no interest to the calling task. The compiler might then be able to replace the task's accept statements with inline procedures guarded as required by low-level event flags. The task, as a separate context, can then be discarded.

Some design tradeoffs must be considered concerning the amount of code executed during a rendezvous. Whether or not a compiler is capable of rendezvous optimizations, it may be more desirable in a particular application to maximize concurrency by minimizing the amount of code executed during a rendezvous. If the code that might be moved into the accept statement is lengthy and has little relevance to the calling task's job, the performance of the calling task may be impaired.

When compiler-provided rendezvous optimizations are unavailable, some techniques can still make rendezvous more efficient for selected tasks of an applications. For example, it may be important that a producer task be able to perform a rendezvous with a consumer task asynchronously, that is, that it need not wait for execution in the consumer task to reach the applicable accept statement. An asynchronous rendezvous can be approximated by positioning a passive buffer task between the producer and consumer tasks. The buffer task accepts calls from either of the

other tasks, stores and retrieves data as requested, and calls no other tasks. Thus the buffer task can spend most of its time being ready to respond immediately to the next caller. Since only minimal processing is performed during the rendezvous (probably only parameter copying), the rendezvous can also be completed quickly. The cost of this technique is an increase in the number of context switches; the benefits are increased concurrency and improved response in critical tasks.

A buffer task is just one form of intermediary task. Others may be useful in a particular application. An active, one-item buffer, often called a transporter task, is designed to call one task to receive data and then to call another task to pass on the received data. Also, a type of intermediary known as a relay task is a hybrid of the buffer and transporter tasks: it both makes and accepts calls to and from other tasks. The problem with a relay task is that any delay it incurs while calling a task may affect its response to incoming calls.

Task arrangements using intermediaries are summarized below. The arrows point from the calling task to the called task, while data are assumed to move from left to right, from the producer to the consumer.

```
Buffer:      (Producer) -> (Buffer) <- (Consumer)

Transporter: (Producer) <- (Transporter) -> (Consumer)

Relay1:      (Producer) -> (Relay) -> (Consumer)

Relay2:      (Producer) <- (Relay) <- (Consumer)
```

A transporter is an active task, a pure caller or actor, while a buffer is a passive task, a pure server. Producer and consumer tasks are either actor or server tasks but never both, while relays are always both and are therefore hybrid tasks. Note that a relay task is equivalent to a back-to-back configuration of a buffer and a transporter task. Actually creating the relay in this manner may help solve its response problem mentioned above, although at the expense of additional context switches. Diagrammed as before, this arrangement appears as follows:

```
Relay1:  (Producer) -> (Buffer) <- (Transporter) -> (Consumer)

Relay2:  (Producer) <- (Transporter) -> (Buffer) <- (Consumer)
```

Another rendezvous efficiency issue concerns the use of the open terminate alternative of the select statement. The presence of the open terminate alternative may add to execution overhead when there are no waiting entry calls because the task is required to determine whether or not it should terminate. It does this by

examining the completion state of other tasks in the current task hierarchy; the more complex the hierarchy, the longer this examination is likely to take.

This overhead can be minimized in several ways, the most obvious of which is to avoid building complex hierarchies of tasks. One can also seek to avoid unnecessary task terminations, especially in embedded systems that perhaps never normally complete execution. If terminations are necessary, the terminate alternative can be guarded so that only the guard expression need be evaluated when the select statement is executed.

The topic of termination overhead is touched upon again below in Section 4.3.1 Binary Semaphores.

Guidelines for Rendezvous Optimization follow.

Efficiency Guideline 2.4-1: Where compiler provided rendezvous optimizations are provided, consider structuring the called task entry to include within the accept statement all code that will be executed as a result of the rendezvous.

Discussion: The compiler may be able to remove simple task interactions and replace them with procedure calls that will permit faster execution.

Efficiency Guideline 2.4-2: To increase application task concurrency and maximize system response, carefully subdivide functions across active and passive tasks, avoiding hybrid tasks if possible.

Discussion: Active tasks should do the main work of the application. Such tasks will probably be the most complex, and they can use calls to other tasks to reduce some of the complexity. Passive tasks are best used to provide controlled access to resources such as storage and I/O devices. The main reason for avoiding hybrid tasks is that their active side exposes them to outside control (and possible indefinite blockage), which may cause failure of their passive functions.

Efficiency Guideline 2.4-3: Where speed of execution is important, it may be wise to avoid building complex hierarchies of tasks. If building such hierarchies is necessary, at least take care to avoid unnecessary task completions and terminations [BAK85].

Discussion: A selective wait statement may include a terminate alternative. The correct implementation of the terminate alternative imposes special demands on the tasking supervisor. It adds execution cost to the selective wait statement that is in the worst case proportional to the number of surrounding levels of nested tasks [BAK85].

Efficiency Guideline 2.4-4: Do not use an open terminate alternative in an intermediary task, such as a buffer task.

Discussion: The purpose of an intermediary task is to speed up rendezvous response, but an open terminate alternative may delay accept processing.

3. Error Recovery

Diagnosis and processing of hardware and software errors are commonly considered among the more difficult problems in the design of embedded real-time applications. Consequently, a reusable part must possess a high degree of reliability and resiliency to error conditions.

Reliability ensures that a part has planned contingencies for all predictable local error conditions so that normal processing is recovered with only minor perturbation to the application. Conversely, resiliency ensures that a part confronted with unpredictable error conditions, for which no planned contingencies are possible, initiates some recovery action possibly outside of the part's control. Relinquishing control of this action may result in a major perturbation to the application. This may be unavoidable in the presence of error conditions that result from exceeding processing capacity limits.

While the application domain warrants a high degree of reliability and resiliency, detecting and accommodating error conditions are frequently restricted by performance efficiency considerations. Therefore, guidelines for error recovery become increasingly important in writing reusable parts. General guidelines that apply to the identification, isolation, and use of exceptions are presented in Chapter 11 of the ARH. However, supplementary guidelines are needed to address specific error conditions that may occur when the application must satisfy real-time requirements. These applications are particularly susceptible to external errors generated by the target execution, deadlock and starvation conditions, storage errors, and timing errors. It is also essential that these conditions be intercepted before they are camouflaged so that recovery action or damage control becomes impossible.

3.1. Reliable Transactions

A transaction between two tasks consists of the exchange of some defined set of information and/or control signals. The mechanisms used by tasks to perform transactions are the rendezvous and accessing shared global variables. The issues involved in designing reliable transactions include ensuring that transactions execute without deadlock, starvation, or thrashing; that transactions occur within the timing constraints imposed by the application requirements; and that exchanged data are protected

from overlapping updates or race conditions. This section focuses on reusability principles that ensure that transactions execute without deadlock, starvation, or thrashing. Timing constraint issues are dealt with in Section 2 Task Scheduling. Data protection issues are dealt with in Section 4.3 Monitors and Semaphores.

Deadlock is a situation in which two or more tasks interact so that none of the tasks can proceed. This state can be caused by contention for resources or by cyclical waiting for services that each provides the other. Starvation is a situation in which one or more tasks are blocked for an arbitrarily long time. In ensuring reliable transactions, only inappropriate starvation resulting from design flaws is of interest. Thrashing is a condition in which tasks fail to rendezvous at a given entry because both caller and called sides of the rendezvous use conditional or timed constructions for calling and accepting the entry.

Since reading or writing global variables cannot produce deadlock or starvation, transactions performed using global variables will be of concern only in those cases in which the rendezvous mechanism is used to protect the global variables, such as through use of a binary semaphore. Use of global variables reduces reusability by creating context dependencies in the program units that access them. Because global variables have other well-known liabilities in addition to this reusability problem, they should be used only to meet performance requirements.

This section also excludes general issues of interprocess communication. Although Ada main programs may exchange data by such means as sharing memory or using network communication facilities, the Ada language provides no direct support for performing safe interprocess communication. In contrast, the rendezvous provides a mechanism for performing safe intertask communication. Since these guidelines deal with Ada language issues, the solutions to problems of providing reliable transactions between Ada main programs lies outside their scope.

The reliability problems dealt with in this section arise from the networks of task interactions that are established when reusable components are composed into applications; they are not inherent in real-time components per se. A component that performs reliably in one composition of an application may not perform reliably in another. This discussion first identifies the kinds of tasks that can appear as reusable components and then discusses problems and methods of application composition. The guidelines for task design are incorporated into the discussion of application composition since specific guidelines for task design respond to specific problems that emerge when composing applications.

3.1.1. Types of Reusable Real-Time Components

Reusable real-time Ada components typically incorporate tasks. Their incorporation of tasks differentiates real-time Ada components from reusable sequential Ada components. Since tasks cannot be independently compiled, reusable real-time components are either packages or separately compiled subprograms. This section discusses two dimensions that characterize real-time components: the actor/server dimension and the atomic/composite dimension.

3.1.1.1. Actor and Server Components

Ada tasks incorporate an intrinsic actor/server model. Pure actor tasks provide no services to other tasks. They may have entry calls for system initialization and shutdown purposes. Pure server tasks have entry calls that provide services to other tasks and do not call on other tasks to provide services. Tasks can combine the attributes of actors and servers, providing services to some tasks while receiving services from others. Such tasks are hybrids.

The actor/server relationship is asymmetric. An actor task must name the server tasks it calls, but server tasks do not know the name of the actor tasks calling them. This asymmetry does not prevent actor tasks from being cast in the form of reusable components; the generic package mechanism provides the means for informing an actor task of the names of entries it is to call. This mechanism is used by declaring, as generic formal parameters, procedures whose parameters match the parameters of the server task entries to be called by the actor task. When the package is instantiated, the generic formal procedure parameters are matched by task entries. The example below illustrates the use of this mechanism to build a system from two server tasks and a generic package containing an actor task that pumps data between the two server tasks.

```
with TBD;
package System_1 is
  task X_Producer is
    entry Give_Me_A (X : out TBD.Data_Type);

  end X_Producer;
end System_1;

with TBD;
package System_2 is
  task X_Consumer is
    entry Here_Is_A (X : in TBD.Data_Type);
  end X_Consumer;
end System_2;
```

```
generic
  type Items is private;
  with procedure Get (I : out Items);
  with procedure Put (I : in  Items);
package Pumps is
end Pumps;
package body Pumps is
  task Move_It;

  task body Move_It is
    I : Items;
  begin
    loop
      Get(I);
      Put(I);
    end loop;
  end Move_It;
end Pumps;

-- This program initiates the execution of a tasking system
-- with a generic actor task that connects two server tasks.

with TBD,
     System_1,
     System_2,
     Pumps;
procedure Main is
  package X_Pump is new Pumps(TBD.Data_Type,
                              System_1.X_Producer.Give_Me_A,

System_2.X_Consumer.Here_Is_A);
begin
  null;
end Main;
```

The skeletal Pumps package shown in this example can be modified to serve a variety of needs such as buffering data, providing for graceful system shutdown, and so forth.

3.1.1.2. Atomic and Composite Components

Atomic components contain only one task. Composite components contain networks of task interactions that may be constructed using reusable components. A composite component must guarantee that its internal network of tasking interactions is not subject to deadlock or starvation.

The information required for incorporating a reusable server component into an application is a specification of the behavior of the task's interface. For an actor component, the information required is a statement of how the component uses the task entries and data types for which it can be instantiated. The

body of a generic package treats generic formal procedure parameters as procedures even if they are instantiated with task entries. Since timed and conditional calls cannot be made to procedures, generic reusable actor tasks make only untimed, unconditional calls on the entries with which they are instantiated.

If a composite component uses resources that are also used by other tasks not part of the component, the composite component may potentially enter a resource contention deadlock with the other task. Therefore, a composite component must identify any shared resource management tasks it accesses so that the potential for resource contention deadlock can be evaluated. This is not a problem for atomic components, since they either are shared resource managers or do not access shared resource managers. If a component accesses a shared resource manager, the component uses two tasks, one of them being the resource manager, and the component is therefore not atomic.

A general principle for constructing composite components is that a pair of composite components should share no tasks except for those managing shared resources. A pair of composite components could share a task object if it is declared in, or accessed through, a package specification used by both composite components. Consider two composite components that have been correctly designed to perform reliable transactions. The reliability of transactions in an individual composite component depends on the interactions among its tasks. If two composite components share a task object L, the safety of each individual component may be compromised by the addition of interactions that were not taken into account when the component was designed. Unless the shared task L manages a common resource required by both composite components, both composite components should use a separate task object to prevent the occurrence of unplanned task interactions.

Providing composite components with separate copies of tasks performing a function common to both composite components can be achieved in two ways: through task types and through generic packages. If the task is directly visible to the composite components, then it should be declared as a task type and each composite component should declare an object of the task type for its own use. Often, reusable components (such as monitors) hide task entry calls inside a procedural interface to perform such functions as enforcing a protocol for calling task entries. In this case, the package enclosing the tasking construct should be generic, and each composite component should declare its own instantiation of the generic package. With either approach, the separate declarations guarantee that the composite components will have no task objects (and therefore no transactions) in common other than those that manage resources.

3.1.2. Corposition of Components

This section describes methods of avoiding deadlock, inadvertent starvation, and thrashing when constructing applications from real-time components.

3.1.2.1. Deadlock

Deadlock occurs when a set of tasks meets two conditions:

1. The tasks are all in a dead state (not making progress through their statements).
2. The tasks depend on each other's progress for exiting from the dead state.

Since none of the tasks is making progress, none of the tasks ever emerges from the dead state. Tasks very commonly enter dead states without being deadlocked. For example, a server task waiting at an accept statement is in a dead state. Deadlocked tasks have not necessarily stopped executing; they may be engaged in some form of busy waiting. Two broad classes of deadlock are those caused by cyclical waiting relationships among tasks (sometimes termed mutual task dependence) and those caused by contention for resources.

3.1.2.1.1 Cyclic Waiting Deadlocks

A cyclical waiting deadlock is one in which a task A is waiting for a task B service, and task B is waiting for a task A service. Since tasks provide each other services through rendezvous, this definition may also be phrased as task A is waiting for a rendezvous with task B and task B is waiting for a rendezvous with task A. A classification of the Ada constructions that can generate cyclical waiting deadlocks is provided by [LEV89]. Two of the important forms of cyclic waiting deadlock are cyclical entry calling and call/wait. In cyclic calling deadlocks, a circular pattern of entry calls is made (A calls B and waits while B calls A and waits). In call/wait deadlocks, A calls an entry in B, B.B1, while B waits for A to call a different entry, B.B2. The call/wait deadlock occurs only if A is the only task that calls B.B2. Either type of deadlock may be direct or indirect.

Both direct and indirect cyclic entry calling are visible in a directed graph representation of the calling relationships among tasks. Call/wait deadlock can occur in any directed acyclic graph of a calling relationship containing at least two paths to one node. Each call made by task A on a different entry of task B constitutes a path in the calling relationship graph.

The following situation illustrates an indirect call/wait deadlock. Task B contains two entries, B1 and B2, in a selective wait. At the time of deadlock, a guard on B1 is closed, so B is waiting on a call to B2. Task A, which has one unguarded entry

A1, is waiting after a call to B.B1. Task C contains calls on B.B2 (which will reset the guard so that task A's call to B.B1 can complete) and on A.A1. If C calls A.A1 before calling B.B2, then A waits on B, B waits on C, and C waits on A. This deadlock does not appear as a cycle in a calling graph, but does appear as two paths to B (AB and CB) in a graph of calling relations among the three tasks.

It is always possible to implement a calling graph containing cycles or multiple paths to one node so that it contains potential cyclic waiting deadlocks. Therefore, if the graph of the calling relationships in an application exhibits the characteristic of showing cycles or having multiple paths to one node, that application must be examined for potential deadlocks.

The easiest way to prevent cyclic task waiting is to organize all task calls into a strict tree hierarchy. Since networks of transactions normally appear in designs, this method is usually too restrictive. In particular, a strict tree hierarchy allows only one task to call a task that manages a resource.

The risk of deadlock caused by cyclic waiting can be reduced by four viable strategies:

1. Convert cyclic calling relationship graphs to hierarchical calling relationship graphs.
2. Establish safe protocols.
3. Establish communication between tasks using well understood buffer and transporter tasking idioms.
4. Examine each cycle for deadlock conditions.

The entire class of circular entry call deadlocks is eliminated by removing cycles from calling relation graphs. This can often be done by changing the direction of an entry call. The following two tasks have a cycle in their calling relationship:

```
task A is
  entry Give_Me_A(Item : in TBD.Data_Type);
end A;

task B is
  entry Give_Me_A(Item : in TBD.Data_Type);
end B;

task body A is
  Local_Item : TBD.Data_Type;
begin
  loop
    B.Give_Me_A(Item);
    accept Give_Me_A(Item : in TBD.Data_Type) do
      Local_Item := Item;
    end Give_Me_A;
    Consume(Local_Item);
```

```
  end loop;
end A;

task body B is
  Local_Item : TBD.Data_Type;
begin
  loop
    accept Give_Me_A (Item : in TBD.Data_Type) do
      Local_Item := Item;
    end Give_Me_A;
    Transform (Local_Item);
    A.Give_Me_A(Local_Item);
  end loop;
end B;
```

As shown below, changing the direction of one call so that A calls B twice eliminates the cycle in the calling relationship.

```
task A;

task B is
  entry Give_Me_A(Item : in  TBD.Data_Type);
  entry Here_Is_A(Item : out TBD.Data_Type);
end B;



task body A is
  Local_Item : TBD.Data_Type;
begin
  loop
    B.Give_Me_A (Item);
    B.Here_Is_A (Local_Item);
    Consume(Local_Item);
  end loop;
end A;

task body B is
  Local_Item : TBD.Data_Type;
begin
  loop
    accept Give_Me_A(Item : in TBD.Data_Type) do
      Local_Item := Item;
    end Give_Me_A;
    Transform (Local_Item);
    accept Here_Is_A (Item : out TBD.Data_Type) do
      Item := Local_Item;
    end Here_Is_A;
  end loop;
end B;
```

Although this transformation removes the possibility of cyclic entry calling, task B could still participate in a call/wait deadlock which would occur if task A called the same entry in task B twice in succession or if A initially called B.Here_Is_A. The possibility of call/wait deadlock can be eliminated by using protocols for safe task interaction. A protocol is a specified pattern of interaction between tasks. For example, tasks A and B in both examples immediately above avoid deadlock by using safe protocols. In the first example, they strictly alternate calls and accepts. In the second example, they order the calls in A and the accepts in B identically. These protocols are very restrictive.

A more generally useful protocol for communication between two tasks is the safe entry calling sequence protocol. Discussion of this protocol is derived from [CEC84]. The protocol requires that for a given sequence of entry calls, an accept statement be reached for a given entry call whenever an accept statement is reached for a previous entry. This ensures that a later entry call will not be permanently blocked by a server waiting at an accept statement for a previously issued entry call. Given a sequence of entry calls made by an actor task, determinating whether the calls are safe depends entirely on the construction of the server.

The examples below illustrate both safe and unsafe calling sequences with a model using an active controller task that calls a server task. Since the safety of the calling sequence depends entirely on the server, only the server is changed to convert an unsafe calling sequence into a safe one.

The controller task passes composite data to the server. The server task processes the components of the composite data and, after processing each component, checks to determine if new data have arrived. If they have, then the server task stops processing the current composite data and accepts delivery of new composite data; otherwise, it continues processing the current data until processing is complete. After an initial delivery of data to the server, the controller task alternates between announcing to the server task that it should accept new data and passing new data to the server task.

```
with TBD;
procedure Shell is
  function The_Data return TBD.Composite_Type is separate;
  task Server is
    entry New_Data_Ready;
    entry Deliver(Data : in TBD.Composite_Type);
  end Server;
  task body Server is separate;
  task Controller is
    entry Take(The_Data : in TBD.Composite_Type);
  end Controller;
```

```
      task body Controller is
        New_Data : TBD.Composite_Type;
      begin
        -- Provide Server with initial data delivery
         accept Take (The_Data : in TBD.Composite_Type) do
           New_Data := The_Data;
         end Take;
         Server.Deliver(New_Data);
         loop
           accept Take(The_Data : in TBD.Composite_Type) do
               New_Data := The_Data;
           end Take;
           Server.New_Data_Ready;
           Server.Deliver(New_Data);
        end loop;
      end Controller;

    begin -- Shell
      loop
        Controller.Take(The_Data);
      end loop;
    end Shell;
```

The sequence of entry calls generated by Controller is

```
   Server.Deliver
   Server.New_Data_Ready
   Server.Deliver
   Server.New_Data_Ready
   ....
```

which contains the two subsequences

```
   Server.Deliver
   Server.New_Data_Ready
```

and

```
  Server.New_Data_Ready
  Server.Deliver.
```

A server body that does not process these subsequences of entry calls safely is shown below.

```
separate(Shell)
task body Server is
  -- Unsafe server
  The_Data : TBD.Composite_Type;
  -- This processing is irrelevant to the safe entry
  -- considerations
  procedure Process (The_Component : in TBD.Data_Type) is
                                                    separate;
begin
```

```
  loop
    accept Deliver (Data : in TBD.Composite_Type) do
      The_Data := Data;
    end Deliver;
    for I in The_Data'Range loop
      Process (The_Data(I));
      select
        accept New_Data_Ready;
        exit;
      else
        null;
      end select;
    end loop; -- through Data_Elements
  end loop;
end Server;
```

In the body of the unsafe Server, the first subsequence of calls, Server.Deliver followed by Server.New_Data_Ready is not safe because the entry Server.New_Data_Ready may not be reached after a call to Server.Deliver. This will happen if Server finishes processing one delivery of data before a new delivery is ready. In this case Controller will wait for Server at New_Data_Ready and Server will wait for controller at Deliver. On the other hand, the sequence of calls Server.New_Data_Ready, Server.Deliver is safe because an accept for Deliver is always reached after an accept of New_Data_Ready. To make the entry calling sequence safe, the body of Server can be modified as shown below.

```
separate(Shell)
task body Server is
-- Safe server

  Signal_Accepted : Boolean; -- initialized in main loop
  The_Data : TBD.Composite_Type;
  procedure Process (The_Component : in TBD.Data_Type) is
separate;
begin
  loop
    Signal_Accepted := False;
    accept Deliver (Data : TBD.Composite_Type) do
      The_Data := Data;
    end Deliver;

    for I in The_Data'Range loop
      Process (The_Data(I));
      select
        accept New_Data_Ready;
        Signal_Accepted := True;
        exit;
      else
        null;
      end select;
    end loop; -- through Data_Elements
```

```
        if not Signal_Accepted then
          accept New_Data_Ready;
        end if;
      end loop;

    end Server;
```

The safe entry sequence protocol can be used to construct reusable components by defining the expected calling sequences for the component's entries and then ensuring that the component will behave safely for those sequences.

Another safe protocol, the priority ceiling protocol, is being developed by The Software Engineering Institute as a real-time scheduling method for Ada. This protocol implements the rate-monotonic scheduling algorithm. A by-product of this scheduling algorithm is that a set of tasks constructed to meet the requirements of the priority ceiling protocol executes free from deadlock. Details of the protocol are provided in [CMUSEI] and [ACM88]. The technical details of the priority ceiling protocol extend well beyond guidelines for writing individual components. For example, successful implementation of the protocol may require modification to Ada runtime executives.

Using standard buffer and transporter idioms as described in [BUH84] and [SHU88] reduces deadlock risk in two ways. First, the idioms remove cycles from graphs of calling relations. Second, the idioms provide protocols that avoid potential for call/wait deadlocks that remains once cycles have been removed from the calling relationship graph. These idioms provide a well-understood method of constructing deadlock-free networks of communicating tasks.

Any design in which the graph of the task calling relations is not a strict tree and which does not use known safe protocols or safe buffer/transporter task communication idioms must be checked for potential deadlocks. Classification of forms of deadlock and methods of altering code to remove potential deadlock situations are presented in [LEV89] and [CHE88]. These references serve as a guide to Ada constructions to be avoided.

3.1.2.1.2 Resource Contention Deadlocks

Resource contention is a form of cyclic waiting in which each task in the deadlock is waiting for a resource controlled by another task in the deadlock. For example, Task A controls resource X and requests resource Y while task B requests resource X and controls resource Y. Four conditions are necessary for resource deadlock to occur:

1. Mutual Exclusion - Tasks must have exclusive access to a resource.

2. Serial Acquisition - Tasks must be able to acquire new resources while holding exclusive access to resources.
3. Non Pre-emption - Tasks holding resources cannot have the resources they control taken away from them.
4. Circular Wait - A cycle must exist in the graph of resource requests and resource grants.

Resource contention deadlock can be prevented by ensuring that at least one of the four conditions for resource contention deadlock is not met. Operating systems literature contains extensive discussion of deadlock avoidance strategies based on preventing the occurrence of each of these conditions. However, many of the approaches deal with overall program design rather than with the construction of reusable components. Therefore, only a brief summary of the approaches is provided.

Mutual exclusion can be prevented only when the nature of the resource allows shared access. For example, disks can be shared, but memory pages in a paged memory system cannot. The choice of preventing deadlock by sharing resources is largely beyond the programmer's control. At best the programmer can avoid managing sharable resources with mutual exclusion.

Strategies for preventing serial acquisition of a class of identical resources are most applicable to building reusable components. Serial acquisition may be prevented if a task can determine in advance the amount of a resource it will use and the resource manager can provide these resources in a block. When the task requests resources, it receives either all the resources it needs or none of them. In either case the requesting task can continue. For example, a task that allocates buffers to a requesting program can be implemented so that it allocates buffers one at a time or in blocks. The latter implementation, combined with a disciplined use of the component in which tasks request all the buffers they will need at once, prevents deadlocks over buffer resources. Although this approach often works well for one class of resource, it has limited utility when it is generalized to a task requesting all resources it needs from all classes of resources at once. The requirement that a task request all its resources at once may prevent the task from ever executing, although it could execute if it acquired and released resources piecemeal. In addition, the all or nothing approach may lead to inefficient use of resources if a task acquires resources at the beginning of a processing cycle that are not used until the end of that cycle.

Non pre-emption can be prevented by writing resource users that pre-empt themselves; that is, they give up resources they control if they cannot obtain resources they need to continue. Although this approach prevents deadlock, self pre-emptive tasks are subject to conditions in which competing tasks never acquire all the resources they need to continue (busy resource contention).

Circular waiting can be prevented by at least two strategies. The first is to divide the resources in a system into classes of identical resources. Each class is then numbered. If all tasks acquire resources within a class in blocks rather than serially, and they acquire resources from different classes in ascending numerical order, then deadlock is prevented. Some task always has control of the resources it needs at the highest resource number allocated. This task can always execute and eventually release the resources. This strategy can be adopted only when it is reasonable for all processes to acquire their resources in the same order. The second strategy for preventing circular waiting is for a resource request manager to maintain a graph of resource requests and grants and refuse to accept any resource requests that will create a cycle in the graph. This strategy typically has a high overhead for managing the resource requests and maintaining and querying the graph.

3.1.2.2. Starvation

Starvation is a condition in which one or more tasks are blocked indefinitely from making progress, but a program continuation may eventually allow the blocked tasks to continue. Starvation resulting from low priority tasks being blocked from execution by high priority tasks is a problem in predictable task scheduling. Task scheduling is dealt with in Section 2 of this annex. In many cases, starvation of a low priority task may be the expected outcome of scheduling decisions made in response to external conditions.

An example of inappropriate task starvation is an actor calling a server when the guard on the server entry is inappropriately closed. The calling task is blocked until the closed guard is opened. A common situation leading to inappropriately closed guards arises when a caller times out while attempting a timed entry call to an entry whose guard employs the count attribute. Consider the case (taken from [BAR84]) of a task that controls access to a variable. The specification of the task requires that the package allow multiple readers but only one writer. In addition, writers are to have priority over readers; that is, as soon as a write entry call is made, no read entry calls are to be accepted. The following select statement is at the heart of the controlling task (updates to the protected variable occur outside the controlling task):

```
loop
  select
    when Write'Count = 0 =>
      accept Start_Read;
      Readers := Readers + 1;
    or accept Stop_Read;
      Readers := Readers - 1;
    or when Readers = 0 =>
      accept Write;
```

```
    end select;
  end loop;
```

Suppose that the select statement evaluates the guards when a writer is queued on the write entry and a context switch occurs before the write entry is accepted. Further, suppose the writer times out before the protection task resumes execution and is removed from the entry queue for Write. If Readers = 0, Start_Read cannot be accepted because the guard is closed, and Stop_Read will not be called because all readers have finished. Therefore, reading tasks will be blocked until a writer calls the open entry for write. If Readers > 0, then reading tasks are blocked at the Start_Read entry until some reader calls Stop_Read. In either case, reader tasks are starved.

A strategy that avoids the starvation problem is for the controlling task to maintain its own counts of readers and writers and to provide a means for writers to decrement the writer count if they time out. The following control loop implements this strategy.

```
loop
  select
    when Writers = 0 =>
      accept Start (S : Service) do
        case S is
          when Read  => Readers := Readers + 1;
          when Write => Writers := Writers + 1;
        end case;
      end Start;
  or
    accept Stop_Read;
    Readers := Readers - 1;
  or
    when Readers = 0 =>
    accept Write;
  or
    accept Stop_Write;
    Writers := 0;
  end select;
end loop;
```

To use this control loop correctly, writers that time out must use a specific protocol to call the controlling task entries. Shown below is the procedure Write, which enforces the required protocol. When the procedure calls Start, it either times out and has no effect on the writer count or the call is accepted and proceeds to call Write. If the procedure times out after calling the Write entry, it decrements the writer count and again leaves the controlling task in a consistent state.

```
procedure Write (X : in Item;
                 T: in Duration;
                 OK : out Boolean) is

  Start_Time : Calendar.Time := Clock;

begin -- Write

  select
    Control.Start(Write);
  or
    Delay T;
    OK := false;
    return;
  end select;
  select
    Control.Write;
  or
    delay T - (Clock - Start_Time);
    Control.Stop_Write;
    OK := false;
    return;
  end select;
  -- update the variable X
  Control.Stop_Write;
  OK := true;
end Write;
```

Two additional variations on this readers/writers problem that provide additional safety measures are presented in Section 3.2 on Fault Tolerance.

3.1.2.3. Thrashing

In the following example of thrashing, conditional select statements are used on both sides of the rendezvous with entry B.B1.

```
task body A is
begin
  loop
    -- Conditional entry call
    select
      B.B1;
    else
      -- Do some processing
    end select;
  end loop;
end A;


task body B is
begin
  loop
```

```
        -- Conditional selective wait
        select
          accept B1;
        else
          -- Do some processing
        end select;
      end loop;
    end B;
```

Unless task A reaches its call to B.B1 at the same time that task B reaches its select statement for B.B1, the two tasks will never rendezvous. In general, the use of timed or conditional selects on both calling and called sides of a given entry should be avoided to prevent this type of synchronization problem.

Guidelines for Reliable Transactions follow.

Composition Orthogonality Guideline 3.1-1: Make tasks in reusable real-time components available as task types or through instantiation of a generic package.

Discussion: If a component provides task objects and is used in the construction of more than one larger component, sharing the task may produce unexpected task interactions leading to deadlock.

Composition Orthogonality Guideline 3.1-2: If possible, organize the task calling relationships in a composite component in a strict tree hierarchy.

Discussion: This organization guarantees that the task interactions in the composite component will not cause cyclic waiting deadlock.

Readability Guideline 3.1-3: Use known safe protocols or well understood buffer/transporter tasking idioms to establish communication between tasks.

Discussion: The potential for producing deadlocks is higher when using ad hoc designs or allowing cycles in the graph of task calling relations. Buffer/transporter communication idioms remove cycles from task calling relations.

Readability Guideline 3.1-4: If a component is designed to be used with a particular safe entry call sequence, document this sequence in the component specification.

Discussion: This documentation is required for the correct use of the component.

Composition Orthogonality Guideline 3.1-5: The design of a resource manager that manages all instances of some class of resources should, if possible, provide multiple instances of the managed resource as a block rather than requiring the user programs to acquire the resources one at a time.

Discussion: This method of allocating resources allows use of a programming discipline that will prevent contention deadlock for the given class of resources.

Composition Orthogonality Guideline 3.1-6: Avoid using the attribute Count in guards of selective wait statements of components.

Discussion: Use of the attribute Count in guards is unreliable if calling tasks time-out or are aborted.

Composition Orthogonality Guideline 3.1-7: If timed entry calls are used with a component that requires use of an entry call protocol to maintain a consistent state, encapsulate the protocol in a procedural interface.

Discussion: Encapsulating the calls to the task entries in a procedure prevents the user from violating the protocols that maintain consistent state information in the server task.

Readability Guideline 3.1-8: If a component performs a conditional or timed accept, indicate this fact in the component specification.

Discussion: A user of the component may never successfully rendezvous with the component if it calls an entry with conditional or timed calls when the entry performs a conditional or timed accept.

3.2. Fault Tolerance

Fault tolerance is a system's capability to provide service despite the faults in one or more hardware or software system components that produce failures. Component faults may be manifested as failure to provide service (e.g., a processor dies) or as provision of erroneous results to other system components (e.g., an implementation of an algorithm produces incorrect results for some set of values).

Providing fault tolerance always imposes the costs of additional processing time, additional hardware, or both. Choosing a system's level of fault tolerance and the method(s) of providing it is a system design issue. Software can play such diverse roles in implementing fault tolerant platforms as managing hardware to provide tolerance of physical component faults (e.g., SIFT, the Software Implemented Fault Tolerant computer [WEN78]),

or managing process rollback and recovery as a means of tolerating transient faults. This section, however, takes the fault-tolerant platform as given. A particular fault-tolerant platform may impose specific constraints on components (e.g., the SIFT computer's executive imposed a specific scheduling algorithm on avionics applications written for it), but there is no generality to the constraints imposed (or resources provided) by various fault tolerant platforms. The focus here, therefore, is on general principles for creating application level reusable components that are tolerant of software faults.

At least two aspects of real-time reusable component design are directly related to providing tolerance of software faults:

1. Guarding against effects of task failure.
2. Detecting and correcting erroneous computations.

Both aspects of design for fault tolerance attempt to localize the effect of software faults. Components designed to guard against task failure are not disrupted by failure of tasks with which they are interacting. Components that are designed to detect and correct their own erroneous computations prevent errors from propagating out of the component.

Distributed Ada programs are Ada main programs that execute on more than one processor. Environments that do not provide transparent fault tolerance for processor failures require distributed Ada programs to manage the process of graceful degradation if one or more of the processors over which the Ada program is distributed fail. This section discusses problems of the semantics of processor failure in such environments.

3.2.1. Tolerance of Task Failure

As discussed in Section 3.1.1 above, real-time components are characterized as containing tasks. The distinctive failure mode of real-time components is the failure of a task in the component. Task failure is defined as a task's becoming unexpectedly abnormal, completed, or terminated. Other component failure modes, such as the propagating of unexpected exceptions by the component, are shared with sequential components and are not dealt with here. An exception being raised during the processing of an accept statement and being propagated into both the calling task and the called task is not considered task failure. Such an exception may lead to task failure if the called task does not handle the exception raised during the accept statement processing. The effects of a task failure are defined in the Ada language, and designing to minimize the effect of such failure is the same whether an Ada program is distributed or is executing on one processor.

For a calling task, the failure of a server task is manifested as the exception Tasking_Error being raised in the calling task at the place of the call [RM 11.5(2)(5)]. This exception must be handled if the calling task is not to terminate. A calling task has two options for handling Tasking_Error: to proceed without the services of the called task or to arrange for the replacement of the called task. Since the choice of options is application-specific, little general guidance is available for calling tasks.

For a called task, the failure of a calling task results in either the entry call's being removed from the queue if rendezvous has not yet occurred or the rendezvous' being completed normally [RM 11.5(6)]. Thus there is no immediate indication that a caller has died. However, it is possible that a calling task must complete a sequence of entry calls to leave a server task in a consistent state. Consistency for a task state may be either internal consistency or consistency with the task's environment. If the calling task can fail without completing the required sequence of entry calls, the called task must guard against being left in an inconsistent state by this failure.

An example of a task that can be left in a state inconsistent with its environment is a monitor task that provides control over the readers and writers of some data structure. Typically, such a task allows multiple concurrent readers but only one concurrent writer, with writers having priority over readers. Such a task may have a state that keeps track of the number of readers to determine when writing is allowed. The following Control task taken from [BAR84] performs this function. The Control task is hidden in a package body whose visible interface consists of the procedures Read and Write.

```
package body Readers_Writers is

  V : Item;

  task body Control is
    Readers : Integer := 0;

  begin
    loop
      select
        accept Start (S : Service) do
          case S is
            when Read =>
              Readers := Readers + 1;
            when Write =>
              while Readers > 0 loop
                accept Stop; -- from readers
                Readers := Readers - 1;
              end loop;
          end case;
        end Start;
```

```
            if Readers = 0 then
              accept Stop; -- from the writer
            end if;
          or
            accept Stop;     -- from a reader
            Readers := Readers - 1;
          end select;
        end loop;
      end Control;

      procedure Read (X : out Item) is
      begin
        Control.Start(Read);
        X := V;
        Control.Stop;
      end Read;

      procedure Write (X : in Item) is
      begin
        Control.Start(Write);
        V := X;
        Control.Stop;
      end Write;

    end Readers_Writer;
```

If a reader is ever aborted after calling Control.Start but before calling Control.Stop, the count of readers will never reach 0, and the state of the task (i.e., the reader count) becomes inconsistent with the actual number of readers in the task's environment. This inconsistency in turn prevents any writes, and, after Start is called by a writer, also prevents any new reads from starting. If a writer is aborted, then the Control task hangs at the accept statement that accepts Stop entry calls from writers and never accepts new readers or writers. Therefore, if a reader or writer could be aborted, the Control task must guard against reader or writer failure.

A way to guard the Control task against reader or writer failure is to employ an agent task to perform the reads or writes. Since the agent task type is declared in the package body, the agent is invisible and cannot be aborted. A package body using the writer agent is shown below. The reader agent would be similar.

```
    package body Readers_Writers is

      V : Item;

      task type Writer_Agent is
        entry Write (X : in Item);
      end Writer_Agent;

      type Writer_Ptr is access Writer_Agent;
```

```
    task body Control is
     --  as before
    end Control;

    task body Writer_Agent is
    begin
      select
        accept Write (X : in Item) do
          Control.Start (Write);
          V := X;
          Control.Stop;
        end Write;
      or
        terminate;
      end select;
    end Writer_Agent;

    procedure Write (X : in Item) is
      WA : Writer_Ptr := new Writer_Agent;
    begin
      WA.Write(X);
    end Write;

    procedure Read (X : out Item) is
    -- ...
    end Read;

  end Readers_Writer;
```

Note that the writer agent is created by an allocator. If it was declared directly as a task object, such as

```
  procedure Write (X : in Item) is
      WA : Writer_Agent;
  begin
     ...
  end Write;
```

then the Writer_Agent would be dependent on the task calling the Write procedure and would be aborted if the task calling Write were aborted. Creation of the agent task with an allocator makes it dependent not on the calling task but on either the package Readers_Writers, if Readers_Writers is a library unit, or on the subprogram or block where Readers_Writers is declared, if Readers_Writers is declared inside another compilation unit [RM 9.4(1-3)].

3.2.2. Tolerance of Software Design Flaws

Hardware faults can be masked by using redundant hardware. The redundant hardware can be configured for use as spares that are swapped in when faults are detected or for simultaneous use as in

the triple modular redundancy (TMR) approach. Unlike hardware, which can physically deteriorate, software malfunctions only if it contains a design flaw. Approaches similar to those used for masking hardware faults can be used to mask faults resulting from undetected software design flaws in fielded systems. The recovery block approach is similar to the use of hardware spares that are swapped in when faults are detected [RAN75]. N-version programming is similar to the TMR approach [AVI85]. Both approaches require multiple implementations of the functionality provided by the computation. They differ in the error-detecting algorithm and in how the execution of the redundant implementations is sequenced. The recovery block uses an application specific test for correctness of execution and executes versions sequentially. The n- version approach executes versions simultaneously and uses a voting algorithm to compare the results of multiple computations, as does the TMR approach to hardware redundancy.

Many variations on the methods for using software redundancy can be created. For example, two computations can be compared; if they disagree, an acceptance test can be applied to determine the correct result. Recovery blocks and n-version programming were chosen to illustrate the use of software redundancy because significant research has been performed with both forms.

Recovery blocks and n-version programming both depend on the assumption that different designs and implementations of the same specification have independent, and therefore different, design faults. Producing multiple versions under conditions that bias the designers and implementers towards creating versions with the same faults destroys the safety margin gained by using multiple versions. Conditions required for achieving independence of design faults is a research topic being pursued at the University of California at Los Angeles [AVI84]. One crucial condition is that specifications be of high quality. If a specification contains flaws, then all designs and implementations based on that specification are biased towards making the
same errors. Other conditions for achieving independence of design faults are discussed in [AVI84]. The successful application of any fault tolerance method using multiple versions of software requires adherence to the conditions for achieving independence of design faults during the production of multiple versions.

3.2.2.1. Recovery Blocks

The basic concept of the recovery block is that the results of a computation are tested and, if the test fails, an alternate implementation of the computation is invoked. If all alternates fail the test, an error signal is returned. A recovery block for performing a sort is shown in pseudo-code form below. DS is the data structure on which the sort is performed:

```
ensure Sorted(DS) and (Sum (DS) = Sum (Prior(DS)) -- Not Ada
  by       Quicksort(DS);
  else by  Mediumsort (DS);
  else by  Sluggishsort (DS);
  else     Error;
end ensure;
```

Here the acceptance test of the sort is that the elements of DS are in sorted order and that the sum of the elements before the sort equals the sum of the elements after the sort. If for some DS the Quicksort fails, then Mediumsort is automatically invoked on the original version of DS. If all alternatives fail, Error is returned. Implementation of a recovery block requires that the state of the system before the invocation of the first alternative be available for use by later alternatives (and possibly for use by the acceptance tests). A recovery block cannot alter the system state until the recovery block verifies that a successful computation has been performed. The recovery block approach to fault tolerance also assumes that the acceptance tests for judging the result of a computation are correct.

These requirements could be met by the following Ada construction (the functions Sorted and Sum are assumed to be correct):

```
with TBD;
package Safesort is
  Sort_Failed : exception;
  procedure Sort (Item : in out TBD.Data_Type);
end Safesort;

package body Safesort is

  -- Three alternate implementations of the computation
  -- functionality
  function QuickSort (Item : TBD.Data_Type)
    return TBD.Data_Type is separate;
  function MediumSort (Item : TBD.Data_Type)
    return TBD.Data_Type is separate;
  function SluggishSort (Item : TBD.Data_Type)
    return TBD.Data_Type is separate;

  -- Functions performing the application specific
  -- acceptance tests
  function Sorted (Item : TBD.Data_Type) return Boolean is
separate;
  function Sum (Item : TBD.Data_Type) return Integer is
separate;

  -- The recovery block which masks design errors in
  -- up to two alternative computations
  procedure Sort (Item : in out TBD.Data_Type) is
     -- Note that the initial state, the in out parameter
     -- Item, is not altered until the final statement
```

```
        -- in Sort.
      Result : TBD.Data_Type;
      type Alternatives is (Quick, Medium, Sluggish);
      Alternative : Alternatives;

    begin -- Sort

      for Alternative in Alternatives loop
        case Alternative  is
          when Quick =>
            Result := Quicksort(Item);
          when Medium =>
            Result := Mediumsort(Item);
          when Sluggish =>
            Result := SluggishSort (Item);
        end case;
        Result_OK := Sorted (Result) and
                     (Sum (Item) = Sum (Result));
        exit when Result_OK;
      end loop;
      if Result_OK then
        Item := Result;
      else
        raise Sort_Failed;
      end if;
    end Sort;
  end Safesort;
```

Note that the initial context is saved and not altered unless a successful sort is performed.

An advantage of the recovery block over n-version programming is that it can recover from errors in n - 1 out of n alternatives. N-version programming using a majority vote decision algorithm can recover from errors in only n/2 - 1 versions. Thus, recovery block provides greater fault tolerance for the same number of versions.

An obvious problem with using the recovery block form for a real-time component is that the worst case execution time is approximately n times worse than best case execution time, where n is the number of alternatives provided. Even the best case execution time always incurs the added overhead of the application-specific acceptance test. These overheads may prevent use of the recovery block format where tight timing requirements exist. The recovery block format may be more useful where fault tolerance is important in code that is not time-critical. Another problem with recovery blocks is that they have no means of recovering from errors in the acceptance tests.

3.2.2.2. N-Version Programming

The basic concept of n-version programming is to execute multiple versions of a required computation and vote on the results simultaneously. Thus, an Ada program that implements the n-version approach to tolerating design faults must be distributed. The most desirable distribution is a processor per version, although acceptable performance may be achieved if execution times of different versions vary significantly and faster versions are grouped together on one processor. The following example shows the n-version form of the safe sort.

```
with TBD;
use TBD; -- to gain visibility to operators

package body Nsort is
  task Quick is
   entry Sort (Item : in TBD.Data_Type);
   entry Result (Q_Result : out TBD.Data_Type);
  end Quick;
  task body Quick is separate;

  task Medium is
    entry Sort (Item : in TBD.Data_Type);
    entry Result (M_Result : out TBD.Data_Type);
  end Medium;
  task body Medium is separate;

  task Sluggish is
    entry Sort (Item : TBD.Data_Type);
    entry Result (S_Result : out  TBD.Data_Type);
  end Sluggish;
  task body Sluggish is separate;

  procedure Sort (Item : in out TBD.Data_Type) is
    Initial_Value : TBD.Data_Type := Item;
    Q_Result, M_Result, S_Result : TBD.Data_Type;
  begin -- Sort
    -- Start concurrent sorts; tasks are distributed
    -- over multiple processors
    Quick.Sort(Initial_Value);
    Medium.Sort(Initial_Value);
    Sluggish.Sort(Initial_Value);

    -- Obtain results
    Quick.Result(Q_Result);
    Medium.Result(M_Result);
    Sluggish.Result(S_Result);

    -- Vote on outcomes
    if Q_Result = M_Result then
      Item := Q_Result;
```

```
    elsif Q_Result = S_Result then
      Item := Q_Result;
    elsif M_Result = S_Result then
      Item := M_Result;
    else -- no majority
      raise Sort_Failed;
    end if;
  end Sort;
end Nsort;
```

Separate successful algorithms can calculate different values that differ by an acceptable range. In that case, the simple test for equality used in the example must be replaced by a test that determines whether the difference in results falls within the acceptable range.

The most important advantage of the n-version form of software fault tolerance is that the execution time of a fully distributed version never exceeds the time for the slowest version plus voting. N-version programming has several additional advantages. The simple majority vote decision algorithm may often be faster (and less likely to contain errors) than the application-specific acceptance tests used by recovery blocks. The work of developing the acceptance tests for each recovery block is eliminated, since essentially the same algorithm is used for all correctness decisions.

The n-version fault tolerance form illustrated above will execute on a uniprocessor system; however, since all versions are executed each time the sort is called, its performance is consistently worse on a uniprocessor than is the performance of the recovery block version.

Unfortunately, implementing n-version fault tolerance inside the Ada language requires a development system that allows specification of the distribution of tasks in an Ada program onto the processors of a target. Because such development systems are not yet commercially available as of 1989, n-version programming can be used on a distributed architecture with current Ada implementations only by stepping outside the Ada language and treating each alternate version as a separate main program.

3.2.3. Semantics of Failure in Distributed Ada Programs

A distributed Ada program is one main program executing on multiple processors. The alternative to distributing Ada programs is designing a distributed application as a group of Ada main programs communicating by facilities not defined by the language. A significant advantage of distributed Ada programs over communicating main programs is that a compiler can check the consistency of the entire distributed application. An application consisting of communicating main programs differs significantly in expressing the method of communication between main programs and

between local tasks within a main program. Communications in a distributed application are all expressed with the same language constructs. This gives an advantage to distributed programs during maintenance, because the location of tasks is transparent to distributed Ada programs. If the distribution of tasks in the application is reconfigured so that local tasks become remote (or vice versa), the distributed application does not require maintenance changes to account for the changed method of communication. Using the same language construct for all task communication also eases the task of reconfiguring a distributed application after processor failure, since resurrecting a failed task on another processor does not change how it communicates with other tasks.

Although the Ada language allows programs to be distributed [RM 9(5)], it does not define the units of distribution or define the semantics of failure for a distributed program. The semantics of failure for a distributed Ada program define the effect of processor failure on the portions of the distributed program executing on the surviving processors. The effects of failure can be broadly divided into two categories: effects on task communication and effects on task contexts [KNI87].

The case of a task engaged in a rendezvous with a task running on another processor that has failed illustrates the lack of definition of failure semantics in Ada. The task on the failed processor is not in a state that is recognized by the language; it is neither elaborated, activated, completed, terminated, or ab- normal. Either of two alternatives is a possible response for a task communication with a task on a failed processor: the calling task could remain in the rendezvous forever or could have Tasking_Error raised at the point of call.

A task referencing a global variable on another processor that has failed illustrates the problem of undefined failure semantics for task contexts.

One approach to eliminating the problem of undefined failure semantics is to design a distributed architecture for extreme physical hardware fault tolerance. The assumption is that the distributed nodes and their communication channels will not fail. This approach is viable for protecting some nodes that perform critical functions, but probably not for the entire distributed target. The space and power requirements of providing physical fault tolerance for all nodes and communication links in a distributed network defeat the reasons for using distributed processing in the first place, such as reducing cabling requirements by localizing processing of sensors and actuators.

The effects of processor failure may be partially controlled by managing the units of distribution of an Ada program. The virtual node unit of distribution discussed in Section 1.1 defines the distributed units so that they never share global data. Such

definition of distribution units increases the fault tolerance of the distributed Ada program by reducing the possible effects of processor failure. In general, the problems produced by the effects of processor failure on task context can be mitigated by defining units of distribution that minimize shared context. The effects of processor failure on task communication between nodes cannot.

Implementers of development systems that allow distribution must either define and implement a semantics of failure or suffer the consequences of undefined program behavior. The second alternative is clearly unacceptable for a production-quality system. A variety of failure semantics has been proposed from which implementers can choose [ACM88].

The issues of the semantics of distributed Ada are being examined by the Ada 9X review process. It is recommended that developers designing distributed programs closely monitor the Ada 9X review process to determine what (if any) semantics for distributed Ada programs will be added to the language.

Guidelines for Fault Tolerance follow.

Composition Orthogonality Guideline 3.2-1: Use agent tasks that are not dependent on a calling task to carry out sequences of entry calls required to maintain an internally consistent state in the called task.

Discussion: This technique ensures that the state of the called task remains consistent if the calling task fails during a sequence of required entry calls.

Efficiency Guideline 3.2-2: If designing components to be tolerant of residual software design faults for a uniprocessor system, consider using the recovery block model.

Discussion: The best case execution time of the recovery block model on a uniprocessor system is significantly better than the n-version model.

RTS Dependency Guideline 3.2-3: If your runtime supports task distribution, consider the n-version programming model for building components that tolerate software design faults with predictable execution times.

Discussion: The execution time of the n-version model is predictable; the execution time of the recovery block model is dependent on whether residual faults are encountered.

Composition Orthogonality Guideline 3.2-4: Use units of distribution for distributed Ada programs that minimize the context shared between units on different processors.

Discussion: This technique minimizes the effects of processor failure on remaining software to effects on task communication.

3.3. Asynchronous Transfer of Control

Asynchronous transfer of control is the capability of one task to cause a change in the execution of another task when the tasks are not synchronized. The task in which the change of execution takes place during asynchronous transfer of control is the "interrupted" task. Tasks are synchronized by performing a rendezvous or by accessing a shared variable to which the pragma Shared has been applied. The only Ada construct that allows limited support for asynchronous transfer of control is the abort statement.

There are at least three situations in which real-time systems need to initiate asynchronous transfer of control: system mode changes, retrieval of partial computational results, and fault recovery [ACM88]. A system mode change occurs when one or more tasks must be stopped or have their flow of control altered in response to an external asynchronous event. Partial computations are performed by tasks whose computations produce an approximate result that is then refined over time to produce a precise result. A task using this result may require the ability to obtain the result at an arbitrary point in the computation. Fault recovery stops or changes a task's execution in response to the occurrence of some fault such as a task's exceeding its time constraints.

At the International Real-Time Ada Workshops (IRTAWs), real-time system experts have proposed several modifications to the Ada language. Each of these modifications was intended to provide a long-range solution to the need for asynchronous transfer of control. At the 1988 IRTAW, the favored solution was to change the Ada language to allow one task to raise an exception in another task [ACM88]. At the 1989 IRTAW, the favored solution was to add a new form of the selective wait statement. Until the completion of the Ada 9X language review process, none of the proposed alternatives will be available. Projects entering the design stage near the end of the Ada 9X review process should track the review process to determine if a mechanism will be provided that enhances Ada's asynchronous transfer of control capability.

In the near term, three alternative approaches can be used to achieve the result of an asynchronous transfer of control: polling, using the abort statement, and stepping outside Ada to control task execution.

The polling approach is designed to meet the need that motivates asynchronous transfer of control. This need arises under the following conditions:

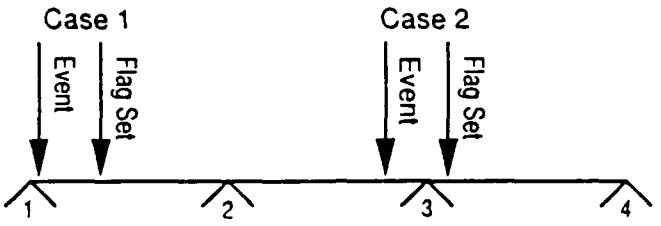
1. A change in execution of the interrupted task must occur within time T1 of the occurrence of some event or system state.

2. The worst case time between the synchronization points in the interrupted task at which a change of execution can take place is T2 (i.e., the task may execute sequentially or wait and sequentially execute for time T2).

3. T2 > T1 (i.e., the time before the task reaches a synchronization point at which it may change its execution may exceed the required response time T1).

The polling strategy requires that the interrupted task check a flag at least every T1 time units for an indication of the occurrence of the event or system state that requires the task to restart or change its control flow. The polling code is interleaved with the normal task processing. Since Ada can implement a variety of scheduling strategies, establishing the points at which polling takes place depends on how the interrupted task is scheduled. In order to select points at which polling code will be inserted, the scheduling strategy must allow determination of an upper bound on the execution time plus waiting time of the interrupted task.

Unless both (1) the occurrence of the event requiring asynchronous communication and (2) setting the flag that the interrupted task is polling are a single atomic event, the design of polling time intervals must take into account the time elapsed between the event and the setting of the flag. In the illustration below, polling occurs at the numbered points on the time line. The time between polling points is T1, the time within which the interrupted task must restart or change its flow of control. In Case 1, the event and the flag setting both occur in the same polling interval; as a result the requirement is met that the interrupted task respond in time T1. In Case 2, the event occurs before polling point 3, and the event flag is set after polling point 3. Thus the time required to respond is T1 plus the time from the event to polling point 3. In this case the interrupted task does not meet the requirement of responding within time T1.



The disadvantages of polling are that it introduces program overhead and significantly reduces the probability that a component will be reusable. An efficient implementation of polling will require the use of shared (i.e., visible to the tasks of interest) variables which in turn introduces context dependencies that must be supplied with each reuse of the component. An additional disadvantage is that the period of polling is fixed by the interleaving of polling code with normal task execution. Since different applications using a polling component typically require different polling periods, the component must be retuned for each application.

The abort statement can be used to emulate restarting a task or changing its flow of control by first aborting a task and then replacing it with either an identical copy or a different task. Either use achieves the goal of an asynchronous transfer of control. The following model for simulating a restartable cyclic task using the abort statement was presented at the 1988 International Workshop on Real-Time Ada Issues as a proof of concept that the abort statement offers at least limited support for asynchronous transfer of control[ACM88].

```
task Background is
  -- Cyclic task that can be asynchronously restarted.

  entry Restart;
  -- POSTCONDITION : the task is restarted at the beginning
  --                 of its loop.

  entry Finish;
  -- For internal calls only.

end Background;

task body Background is
...

begin
```

```
      ...
      loop
        declare
          task Computing_Task;
          task body Computing_Task is
          ...
          begin
            Compute;
            Background.Finish;
          end Computing_Task;
        begin
          select
             accept Restart;
             abort Computing_Task;
          or
             accept Finish;
             exit;
          end select;
        end;
      end loop;
    end Background;
```

This skeleton provides for the computing task and its enclosing controlling task to terminate normally, if the computation task finishes, or to have the computing task restarted, if the restart entry of the controlling task is called. The skeleton can be easily modified to provide for mode-changing emulation where a different task is started after a mode change.

Using the abort statement may result in at least three significant problems that may prevent its being a serious candidate for implementation of asynchronous transfer of control:

1. Both the abort statement and the creation of new tasks to replace the aborted tasks have high overhead

2. There is no guarantee that storage used for aborted tasks will be reclaimed

3. The language definition does not guarantee that aborted tasks will terminate if they enter an infinite loop that does not contain synchronization points for the abort statement.

If both polling and the abort statement can be used to produce a component allowing asynchronous transfer of control (i.e., if both meet timing requirements and the abort statement does not generate memory problems), then the component produced using the abort statement will have a higher probability of reuse: the timing of the asynchronous transfer of control is not hardwired into the component as it is with a component using polling. However, the runtime behavior of the system will probably dictate the choice of method for implementing or simulating asynchronous

transfer of control. For example, a long-running system that does not reclaim storage for aborted tasks cannot use the abort statement to construct components providing asynchronous transfer of control because using the abort statement exhausts memory.

If neither polling nor the abort statement provides the required response, the radical approach of directly manipulating the run-time representation of tasks can be used. For example, one project has successfully restarted tasks by using assembly language to reset their stack and program counters. This approach should be used only as a last resort, since the possibility of producing unexpected program behavior is very high. If may also be difficult to apply these techniques without violating Ada semantics. What, for example, is the effect of resetting the program counter and stack pointer of a server task engaged in a rendezvous?

Guidelines for Asynchronous Transfer of Control follow.

Transportability Guideline 3.3-1: Use polling or abort constructs before stepping outside the Ada language.

Discussion: These methods of achieving the goals of asynchronous transfer of control are safer than direct manipulation of a task's representation in the runtime system.

Transportability Guideline 3.3-2: Use the pragma Shared to synchronize access to shared global flags used for the polling solution to asynchronous transfer of control problem.

Discussion: The language definition requires the use of the pragma Shared for shared variables accessed outside rendezvous. Note that the pragma Shared simply ensures that each time a task uses a shared variable it will work with the latest value of that variable. Certain compiler optimizations that make it possible for a task to work with an obsolete value of a shared variable are prevented by the pragma Shared.

RTS Dependency Guideline 3.3-3: Write all tasks that may be aborted so that they contain an abort synchronization point inside all loops that have the potential for executing indefinitely.

Discussion: The language definition allows both asynchronous and synchronous implementations of the abort statement. The asynchronous abort stops a task at any point. The synchronous abort stops a task only at abort synchronization points [RM 9.10(5-6)]. There are two methods of inserting abortion synchronization points. The first is to insert delay statements with a wait of 0.0. This method may have undesirable side-effects depending on the implementation of the runtime system, e.g., task rescheduling. The second is for a task to apply the

attribute Callable to itself to determine if it has been aborted and to exit the loop if it has. This method can be used by incorporating the attribute Callable into a loop condition or exit statement or by enclosing a call to an abortion synchronization point inside an if statement. The three constructions below illustrate the use of the attribute Callable to prevent an aborted task from looping infinitely.

```
while Some_Condition and Interrupted_Task'Callable loop
   -- Some processing
end loop;

loop
   -- Some processing
   exit when not Interrupted_Task'Callable;
end loop;

loop
   -- Some processing
   if not Interrupted_Task'Callable then
     abort Interrupted_Task;
   end if;
end loop;
```

An aborted task reaching an abort synchronization point immediately becomes completed. Since the statement inside the if in the third construction is an abort synchronization point, the task becomes completed as soon as it reaches the statement, and the statement is never executed. Therefore, it does not matter what abortion synchronization statement is used [RM 9.10(6)].

The idioms using the attribute Callable prevent the possible undesirable side effects of using the statement "delay 0.0". Therefore, components constructed using the idioms employing the attribute Callable are more reusable. The statement "delay 0.0" should be used only when it meets timing constraints that the idioms using the attribute Callable fail to meet.

RTS Dependency Guideline 3.3-4: Document assumptions made about the Ada run time when the abort statement is used to achieve asynchronous transfer of control in the interface specification of the component.

Discussion: The reusability of a component using the abort statement depends on the assumptions made about the runtime executive when the component was implemented. For example, the component may assume that the runtime executive reclaims space allocated to aborted tasks. If the component is used with a runtime that does not support this assumption then unexpected storage errors may occur when the component is reused.

4. Resource Control

4.1. Task Identification

In the development of reusable parts that dynamically control resources under real-time constraints it is often necessary for a part to be able to have some way of identifying the task that encloses its execution. This identity may be used to locate resources assigned to the part or to be passed as an entry call parameter to another task that may perform some specific service based upon this identity. For example, in the earlier paradigm for changing task priorities, it was necessary for a task to be able to identify itself.

One method of obtaining this identity is for a task to request it from the task responsible for its activation, i.e., the parent or creator task. This method usually requires that the two tasks rendezvous so that the parent task can notify the newly activated task of its identity. Normally, for tasks of the same type, the identity is an index into an array of task objects. Unfortunately, this method may be unsatisfactory in highly parallel execution environments because each task cannot proceed with its execution until the rendezvous with its parent task has been completed. To minimize the time each task is blocked because of the enforced serialized execution, alternate approaches should be considered that are reusable in both serial and parallel execution environments. Use of such alternatives requires that the effect of performing the rendezvous be accomplished in some other fashion. For example, one approach is illustrated in the following reusable part where the identity of a task is established as a result of its activation and is therefore immediately available upon executing the task.

```
declare
  Task_Set_Size : constant := ...;
  type Reusable_Task_Set_Type is range 1.. Task_Set_Size;
  -- Task type requiring identification.
  task type Reusable_Task_Type;
  type Reusable_Task_Pointer_Type
       is access Reusable_Task_Type;
  type Reusable_Task_Array_Type
       is array (Reusable_Task_Set_Type)
                 of Reusable_Task_Pointer_Type;

  Reusable Tasks : Reusable_Task_Array_Type;

  task body Reusable_Task_Type is
    -- Individual task identification established
    -- as a result of task activation.
    Task_Id : constant Reusable_Task_Set_Type
            := Global_Task_Id;
  begin
    ...
```

```
    end Reusable_Task_Type;
    ...
begin

    for Task_Id in Reusable_Tasks'Range loop
      -- Assertion: Global_Task_Id is safe.
      Global_Task_Id           := Task_Id;
      Reusable_Tasks(Task_Id) := new Reusable_Task_Type;
    end loop;
    ...
end;
```

In the above example a set of tasks of the same task type are referenced through an array of access types that designate the task objects. The approach relies upon the identity task and its parent having access to a global object, Global_Task_Id, that is a value of the index type of the array. The example does not show Global_Task_Id, which is assumed to be visible to both tasks. The value of this global object is maintained by the parent task as the identification of the next task that it activates as a result of evaluating the allocator. Since the allocator activates the identity task, elaboration of the task's declarative part establishes a constant set to the current value of this global object. As a consequence, the identity of the task is available immediately upon execution. Using this approach, each task may commence execution without having to be queued for service by its parent task. An informal assertion is made that the global object is safe on the premise that it is modified only within the loop and that the activation of the identity is serialized as required by the RM for the evaluation of the allocator.

This example uses a global object to present the technique simply and efficiently for real-time applications. Unfortunately, there is a flaw in using a simple global object: a task could masquerade as another task by using, accidentally or deliberately, a different task identity. This situation might occur if the global object was referenced after the task had begun execution and the parent task had activated additional tasks. This flaw can be rectified by securing Global_Task_Id within the private part of a package that declares Task_Id_Type as limited private.

```
package Secure_Task_Id Package is

  type Task_Id_Type is limited private;

private

  Global_Task_Id : Reusable_Task_Set_Type;
  Null_Task_Id   : constant Reusable_Task_Set_Type := ...;
  type Task_Id_Type is
    record
```

```
        Task_Id : Reusable_Task_Set_Type
                  := Global_Task_Id;
    end record;

end Secure_Task_Id_Package;
```

The use of this package by a task would require that Task_Id be declared as a variable. This technique is safe, since the task would not be able to change the variable since its type is limited. However, to thwart a subsequent declaration that would yield a different value from that resulting from task activation, Global_Task_Id should be assigned a null value prior to the tasks commencing execution. When the priority of the parent task is greater than the created tasks, the parent may explicitly perform the assignment as was shown in the paradigm for changing task priority. Alternatively, a safer method is for the default assignment in the type declaration of Task_Id Type to use a function that before returning the value of Global_Task_Id resets it to the null value. The type declaration and function would then be defined as follows:

```
function Set_Task_Id return Reusable_Task_Set_Type is
  Local_Task_Id  : Reusable_Task_Set_Type
                   := Global_Task_Id;

begin
  Global_Task_Id := Null_Task_Id;
  return Local_Task_Id;
end Set_Task_Id;

type Task_Id_Type is
  record
    Task_Id : Reusable_Task_Set_Type
              := Set_Task_Id;
  end record;
```

Between the time of the assignment to the global object in the parent task and its subsequent reference in the identity task, another task may be able to intercept the value by executing in parallel. If so, the global object must be protected through some synchronization facility. This requirement may detract from the efficiency of the technique.

Finally, when tasks of different types are to be identified, a similar technique may be used. In this instance, a more complex type of identification must be devised, such as the index value combined with an access value that designates the task access type array.

4.2. Interrupts

Endemic to real-time embedded applications is the requirement to respond to external conditions or interrupts. The response usually initiates processing of the interrupt, and in many instances, the suspension of current processing, such as when processing is interleaved on a single computer. In writing reusable software, this requirement raises two important issues: how to minimize the time to process the interrupt and how to ensure that there is no deleterious effect from asynchronously suspending the currently executing task. It is not possible to resolve these issues while safeguarding all the commonly understood tenets of reusability. Consequently, a tradeoff is necessary in order to develop guidelines that may rely on some degree of cooperation from the underlying Ada Virtual Machine.

The minimization of interrupt processing time, while dependent upon the idiosyncrasies of the runtime system, may be achieved by writing interrupt handling software in a manner that is conducive to its direct execution [ACM88b]. Direct execution allows the task enclosing the interrupt handler to be executed without the usual overhead of changing task context. This result is achieved through specific restrictions on the construction of the interrupt handler (the accept body that is bound to the interrupt entry) and the assumption that the execution of the handler will not be blocked or suspended unless it becomes necessary due to the occurrence of another interrupt of higher priority. Optimally, the accept body should minimize references to local objects and subprogram calls; it should not precipitate the occurrence of a synchronization point. When processing associated with the interrupt can be deferred, the accept body may include a call to a trivial entry.

The notion of a trivial entry in writing interrupt handlers as "signalling interrupt tasks" [ACM87a] may improve the reusability of interrupt processing software. This improvement results because interrupt processing software execution time is functionally distributed in order to reduce the probability that interrupts are lost and to enable the resumption of any suspended task according to its priority. Since the accept statement for a trivial entry does not include an accept body, the enclosing task is at the accept statement the client task need not be suspended. The only required action by the RTS is to update the entry queue and to unblock the task enclosing the trivial entry. When the client task is an interrupt handler, the task may then continue execution, thereby reaching the interrupt entry with a minimum of processing overhead. If no new interrupts are pending, any task blocked by the execution of the interrupt handler may be executed according to its priority. To ensure that the interrupt handler is not suspended because the called task is not waiting at the accept statement for the trivial entry, it is recommended prac-

tice to use a conditional entry call in the interrupt handler. The following example illustrates the use of a trivial entry in writing reusable interrupt processing software:

```
with SYSTEM;
generic
  Int_Address : SYSTEM.Address;
  with procedure Contingency_Processing;
  with procedure Process_Interrupt;
package Reusable_IHP_Package_Template is

  task Interrupt_Signalling_Task is
    entry Trap_Interrupt;
    for Trap_Interrupt use at Int_Address;
    pragma Priority (SYSTEM.Priority'Last);
  end Interrupt_Signalling_Task;



  task Interrupt_Processing_Task is
    -- Assertion: Trivial entry
    entry Forward_Interrupt;
    pragma Priority (SYSTEM.Priority'Last-1);
  end Interrupt_Processing_Task;

end Reusable_IHP_Package_Template;

package body Reusable_IHP_Package_Template is

  task body Interrupt_Signalling_Task is
  begin
    loop
      -- Real-Time Assertion:
      -- Loop execution can only become blocked due
      -- to the delay between interrupts.
      accept Trap_Interrupt do
        select
          Interrupt_Processing_Task.Forward_Interrupt;
        else
          Contingency_Processing;
        end select;
      end Trap_Interrupt;
    end loop;
  end Interrupt_Signalling_Task;

  task body Interrupt_Processing_Task is
  begin
    loop
      accept Forward_Interrupt;
      Process_Interrupt;
    end loop;
  end Interrupt_Processing_Task;
```

```
    end Reusable_IHP_Package_Template;
```

Guidelines for Interrupts follow.

RTS Dependency Guideline 4.2-1: Document the use of trivial entry calls.

Discussion: The reusability of interrupt processing software may depend upon the use trivial entries. When there is an explicit dependency by a reusable part for the RTS to recognize and support trivial entries, it must be documented.

RTS Dependency Guideline 4.2-2: Interrupt entries should ensure that no dependence is placed upon the continuation of the hardware task priority outside of the accept statement when the associated entry is called as a result of a hardware interrupt.

Discussion: Typically, interrupt handlers are enclosed in a loop statement. A reusable part for an interrupt handler should not depend upon the continuation of the loop at the same priority beyond the first synchronization point following completion of the accept body. While some implementations may raise all interrupt processing to that of the hardware task priority, this approach is inconsistent with the RM.

RTS Dependency Guideline 4.2-3: Avoid using a terminate alternative in a selective wait statement that encloses an accept alternative for an interrupt entry.

Discussion: Depending upon the implementation, the use of a terminate alternative and an accept statement for an interrupt entry in the same selective wait statement may prevent the termination of a program. This effect is a result of an entry call's being issued by a hardware task that is not required to meet the conditions required for the terminate alternative to be selected.

RTS Dependency Guideline 4.2-4: The task specification for an interrupt handler should only declare a single entry and the interrupt to which it is bound.

Discussion: The reusability of an interrupt handler may depend upon minimizing interrupt processing time. Therefore, to promote direct execution of the accept body of the entry bound to an interrupt, a task specification should only declare a single entry and the interrupt to which it is bound.

RTS Dependency Guideline 4.2-5: The task body for an interrupt handler should contain only the accept body for the interrupt entry enclosed in a loop.

Discussion: The reusability of an interrupt handler may depend upon minimizing interrupt processing time. Therefore, to promote direct execution of the accept body of the entry bound to an in-

terrupt, a task body should contain only the accept body for the interrupt enclosed in a loop without an associated iteration scheme. However, using this technique does not relax the observance of Guideline 4.2-2.

RTS Dependency Guideline 4.2-6: An interrupt entry should be called only by a hardware task.

Discussion: The reusability of an interrupt handler may depend upon minimizing interrupt processing time. Therefore, to promote direct execution of the accept body of the entry bound to an interrupt, the entry should only be called as a result of an interrupt, i.e., a hardware task.

4.3. Monitors and Semaphores

4.3.1. Binary Semaphores

A semaphore is a form of entry "counter" used to control the number of concurrently active execution threads in a critical code section or other system resources. The counter value is modified using the so-called P and V operations (hereafter referred to as "seize" and "release" operations respectively) to record entry into and departure from the critical section. Presumably, a binary semaphore would limit the value of the counter to "0" and "1" as a way to guarantee that the critical section code is executed by only one thread at a time.

A literal implementation of this concept might use an integer shared variable that is tested and incremented, or decremented, as required. The obvious problem with this approach is that the shared integer itself becomes "critical" if no fool-proof provision exists for blocking and queuing simultaneous, or nearly simultaneous, attempts to reach the critical section. This need to block and queue differentiates the semaphore from the more elementary event flag. A simple Ada solution employs a rendezvous with a semaphore task such as the following:

```
task Binary_Semaphore is
  entry Seize;
  entry Release;
end Binary_Semaphore;

task body Binary_Semaphore is
begin
  loop
    accept Seize;     -- seize critical section
    accept Release;   -- release critical section
  end loop;
end Binary_Semaphore;
```

Upon initiation, the Binary_Semaphore task immediately blocks at the first accept statement inside the loop, waiting for a call to the Seize entry. When the call arrives, a very brief rendezvous occurs, after which the Binary_Semaphore task blocks at its second accept statement. Any further calls to the Seize entry block (and queue up) until the task receives a call to the Release entry, completes the rendezvous, and loops back again to the first accept. Thus, used properly, this task provides the blocking and queuing needed to coordinate access to a critical section.

The description above illustrates operation of an initially "unlocked" binary semaphore. The binary semaphore can be used in the opposite manner by calling the Release entry first. The semaphore then appears initially "locked," blocking the caller until a Seize entry call causes back-to-back rendezvous and recycles the semaphore.

One practical element is missing from the Binary_Semaphore task: it provides no means for terminating the task. The desirability of being able to terminate the task in a real-time system is likely to be application-dependent. One might ask, for example, if the system will ever run to completion or simply run until the processor fails. If termination is required, one might consider effecting it as follows:

```
task Binary_Semaphore is
  entry Seize;
  entry Release;
  entry Stop;
end Binary_Semaphore;

task body Binary_Semaphore is
begin
  loop
    select
      accept Seize;  -- seize critical section
    or
      accept Stop;
      exit;
    end select;
    select
      accept Release;  -- release critical section
    or
      accept Stop;
      exit;
    end select;
  end loop;
end Binary_Semaphore;
```

This approach would certainly work, but with a possibly undesired side effect. A call to the Stop entry forces an immediate exit from the loop and completes the execution of the task without

processing entry calls that may be queued at an entry controlled by the other select. The loss of the entry calls (and the raising of the TASKING_ERROR exception in the callers) can be avoided by using an open terminate alternative in the select statement. This approach, however, does cost some extra execution overhead during each rendezvous [BUR86]. A version that uses an open terminate alternative would look like this:

```
task Binary_Semaphore is
  entry Seize;
  entry Release;
end Binary_Semaphore;

task body Binary_Semaphore is
begin
  loop
    select
      accept Seize;  -- seize critical section
    or
      terminate;
    end select;
    select
      accept Release;  -- release critical section
    or
      terminate;
    end select;
  end loop;
end Binary_Semaphore;
```

Because of the overhead incurred, this approach is not recommended. In addition, whether or not the Binary_Semaphore task should be stopped with the critical section still unreleased, as the second select permits, is problematical and depends on the needs of the application.

During the development of the Binary_Semaphore code, it may become desirable to encapsulate the task in a package, perhaps for reasons of reusability. However, if such a package is provided as a library package, the execution behavior of the task may be affected. Ada does not define how such tasks terminate; the execution of the terminate is implementation-dependent. Therefore, the open terminate alternative form of the Binary_Semaphore could terminate unpredictably any time after elaboration. An altered version that provides a Stop entry to invoke the terminate alternative has the virtue of at least postponing the uncertainty until after the task is told explicitly to terminate. It also prevents incurring the additional select statement execution overhead (imposed by the open terminate alternative [BAK85]) during normal seize and release processing. An even better approach is to have the package export a Binary_Semaphore task type rather than the task itself so that the task object may reside in a construct other than a library package. Such a version is presented in Guideline 1.4.3-1 below.

The implementation of a binary semaphore is straightforward and simple to use, but it may not be suitable for every real-time application. For example, it is not clear that it will execute quickly enough due to the context switching overhead it incurs. However, a sufficiently sophisticated compiler may be able to remove the task call through (Habermann-Nassi) optimization and replace it with in-line references to one or more operating system event flags, thereby eliminating the context switch.

Guidelines for Binary Semaphores follow.

Efficiency Guideline 4.3-1: If termination of a binary semaphore task is a requirement, use an explicit termination entry point rather than an open terminate alternative.

Discussion: An open terminate alternative incurs additional overhead each time the enclosing select statement is executed. The following example employs an explicit termination entry point in a binary semaphore task type that is exported by a package. By exporting a type rather than the task itself, the task object may be created in a suitable master construct and better control over task termination may be achieved.

```
package Binary_Semaphore_Package is
  task type Binary_Semaphore_Type is
    entry Seize;
    entry Release;
    entry Stop;
  end Binary_Semaphore_Type;
end Binary_Semaphore_Package;

package body Binary_Semaphore_Package is
  task body Binary_Semaphore_Type is
    Stopped : Boolean := false;
  begin
    loop
      select
        accept Seize;  -- seize critical section
      or
        accept Stop;
        Stopped := true;
      or
        when Stopped =>
          terminate;
      end select;
      accept Release;  -- release critical section
    end loop;
  end Binary_Semaphore_Type;
end Binary_Semaphore_Package;
```

Efficiency Guideline 4.3-2: Consider avoiding the use of binary semaphores except in an application that will be developed using a compiler capable of optimizing simple tasks completely out of the executable code.

Discussion: A high-speed application with very tight timing requirements may not be able to use a binary semaphore because of the overhead incurred when the semaphore remains a separate task in the final executable image.

Reliability Guideline 4.3-3: Encapsulate the binary semaphore and the critical section in a package that exports a procedure that provides the only way to access the semaphore and the code it guards.

Discussion: A package of this sort is actually a form of monitor, which should be the preferred form of protection for a critical section. The implicit suggestion here is that a binary semaphore used by itself is not the safest way to proceed since its use becomes, in effect, voluntary. A task or subprogram that is supposed to call the semaphore may "forget" to do so and then execute the critical section anyway. Worst of all, the application may just happen to work correctly during testing, hiding its fatal flaw until too late. The encapsulated semaphore might appear as follows:

```
package Critical_Section_Package is
  procedure Critical_Section;
end Critical_Section_Package;

with Binary_Semaphore_Package;
package body Critical_Section_Package is

  package BSP renames Binary_Semaphore_Package;
  Binary_Semaphore : BSP.Binary_Semaphore_Type

  procedure Critical_Section is
  begin
    Binary_Semaphore.Seize;
    --
    -- critical section code
    --
    Binary_Semaphore.Release;
  end;
end Critical_Section_Package;
```

4.3.2. General Semaphores

A general semaphore provides controlled access to a group of related resources. It keeps a count of available resources, decrements the count for each access request, and then blocks and

queues any access requests that arrive after all the resources have been allocated. When a resource is released, the next call on the entry queue is processed and allowed access to the resource.

A general semaphore requires three constructs. First is a set of two procedures that provide the Seize and Release capability that the client tasks use to make their access requests. Second is a binary semaphore used to ensure mutual- exclusion execution of the critical (resource count manipulation) code in the two procedures. Third is another binary semaphore that is used to buffer requests that must wait because the requested resource is exhausted.

Buffering at a binary semaphore is accomplished by queuing request calls at the Release entry while the semaphore task waits for rendezvous at its Seize entry. This is the initially "locked" mode of operation. Thereafter, as long as there are requests in the queue, the Seize entry is called each time a client task releases resources. This call causes a Seize rendezvous, a Release rendezvous that releases the next access request, and a recycling of the semaphore task to block again at the Seize entry.

For the following example, assume a package Binary_Semaphore_Package that exports a task type Binary_Semaphore_Type, and a package Global_Data that exports resource counter R_Count (initialized to some positive value).

```
package General_Semaphore_Package is
  procedure Seize;
  procedure Release;
end General_Semaphore_Package;

with Binary_Semaphore_Package;
with Global_Data;
package body General_Semaphore_Package is

  package B_Sema renames Binary_Semaphore_Package;

  -- mutual exclusion semaphore
  Sema_Mutex  : B_Sema.Binary_Semaphore_Type;
  -- buffer semaphore
  Sema_Buffer : B_Sema.Binary_Semaphore_Type;
  procedure Enqueue_Request renames Sema_Buffer.Release;
  procedure Dequeue_Request renames Sema_Buffer.Seize;

  procedure Seize is          -- resource request
    begin
      Sema_Mutex.Seize;                    -- seize counter
      Global_Data.R_Count := Global_Data.R_Count - 1;
      if Global_Data.R_Count < 0 then      -- resource exhausted
        Sema_Mutex.Release;                -- release counter
```

```
          Enqueue_Request;                    -- queue request
        end if;

        Sema_Mutex.Release;                   -- release counter
      end Seize;

    procedure Release is          -- resource release
      begin
        Sema_Mutex.Seize;                     -- seize counter
        Global_Data.R_Count := Global_Data.R_Count + 1;
        if Global_Data.R_Count <= 0 then      -- request is waiting
          Dequeue_Request;                    -- de-queue request
        end if;
        Sema_Mutex.Release;                   -- release counter
      end Release;

  end General_Semaphore_Package;
```

Guidelines for General Semaphores follow.

Efficiency Guideline 4.3-4: Analyze timing considerations when proposing the use of a general semaphore and design to minimize the occurrence, if possible, of requests for a resource when the resource supply is exhausted.

Discussion: In the absence of the proposed Habermann and Nassi rendezvous optimization, the general semaphore provides a possibly expensive (in execution overhead) capability for controlling access to shared resources. The mechanism requires four context switches when resources are maximally allocated, possibly the very time when processor cycles are most in demand. Otherwise, just as with the binary semaphore, only two context switches are required when a resource is immediately available.

Reliability Guideline 4.3-5: Where use of a general semaphore is indicated, consider substituting a monitor.

Discussion: A basic general semaphore used by itself has the same enforcement limitation as a binary semaphore in that its use is "voluntary" and is effective only if it is called in every instance in which it is required. The use of the semaphore function can be enforced by packaging it together with the resource being requested. The resulting structure will be a monitor that accomplishes the same goal much more safely.

4.3.3. Monitors

A monitor is a program construct that enforces a protocol of using predefined procedures for access to shared data. The basic feature of a monitor is that it includes (or encapsulates) a collection of shared data and the procedures that access the data.

The critical sections are located within the procedures controlled by the monitor. The code is designed so that only one process may be executing in the monitor (i.e., executing one of the procedures) at any one time. This arrangement ensures that the shared data are neither accessed outside the monitor nor simultaneously. Hence mutual exclusion is ensured. The monitor was proposed by Brinch Hansen [BRI73] and further developed by Hoare [HOA74] [Shumate, p.39].

The following is an imaginary monitor construct written in Ada-like syntax:

```
monitor Item_Section is     -- ! this is not Ada !

  Item : Integer;

  procedure Add (Amount : in Integer) is
  begin
    Item := Item + Amount;
  end Add;

begin      -- initialization of the monitor
  Item := 17;
end Item_Section;
```

This monitor includes the implicit provision that calls are serialized so that the procedure Add is executed indivisibly on behalf of each caller. The initialization portion of the monitor is executed only once--before the monitor accepts any calls to the monitor procedure. The result is that Item is properly incremented even by multiple simultaneous calls to Add, because the monitor ensures that the calls are executed serially and indivisibly [Shumate, pp. 39-40].

A real Ada monitor can be built as a package of visible procedures together with a hidden task that encapsulates the shared data. The hidden task is simply an expansion of a binary semaphore that includes a sequence of statements effecting some manipulation of shared data during the rendezvous. The following is an example of an Ada monitor package [Shumate, p.65].

```
package Monitor is
  procedure Add_Data (Amount : in Integer);
  procedure Read_Data (Total : out Integer);
  procedure Stop_Monitor;
  pragma Inline (Add_Data, Read_Data);
end Monitor;

package body Monitor is

  task Mutex is
    entry Add_Data (Amount : in Integer);
    entry Read_Data (Total : out Integer);
```

```
      entry Stop;
    end Mutex;

    task body Mutex is
      Sum     : Integer;
      Stopped : Boolean;
    begin
      Sum := 0;
      loop
        select
          accept Add_Data (Amount : in Integer) do
            Sum := Sum + Amount;
          end Add_Data;
        or
          accept Read_Data (Total : out Integer) do
            Total := Sum;
          end Read_Data;
        or
          accept Stop;
          Stopped := true;
        or
          when Stopped =>
            terminate;
        end select;
      end loop;
    end Mutex;

    procedure Add_Data (Amount : in Integer) is
    begin
      Mutex.Add_Data (Amount);
    end Add_Data;

    procedure Read_Data (Total : out Integer) is
    begin
      Mutex.Read_Data (Total);
    end Read_Data;

    procedure Stop_Monitor is

    begin
      Mutex.Stop;
    end Stop_Monitor;

  end Monitor;
```

The interface to the shared data is through the procedures in the package specification. They are "Inline" in order to avoid the overhead of a procedure call. The package body reveals that they do nothing but call the appropriate entries in the task "Mutex" (for MUTual EXclusion). Such procedures are called "entrance procedures" [Shumate, p.66].

Guidelines for Monitors follow.

Efficiency Guideline 4.3-6: Use pragma Inline (assuming it is appropriately supported in the implementation being used) for the entrance procedures of a monitor.

Discussion: This optimization avoids incurring the overhead of a procedure call on the part of the calling subprogram or task.

Reliability Guideline 4.3-7: Use a monitor construct instead of a binary semaphore to control multiple simultaneous access to shared data.

Discussion: The monitor encapsulates not just the critical sections of code that manipulate the shared data, but also the data itself. Therefore, disciplined access to the data must be observed. The protocol is not voluntary, eliminating the chances of sidestepping it accidentally.

Reliability Guideline 4.3-8: Use a monitor construct instead of a general semaphore to control multiple simultaneous access to a resource.

Discussion: With the use of the pragma Inline (assuming it is appropriately supported in the implementation being used), the monitor form of the general semaphore incurs no more overhead than a general semaphore, i.e. up to four context switches, but affords greater protection.

4.4. Storage Reclamation and Reuse

One of the constraints placed on real-time systems is the requirement to meet time deadlines. For embedded systems, the constraint of limited memory capacity is also likely to be imposed. This constraint means that the available capacity must be carefully managed so that there will always be sufficient memory to accommodate the application code and static data, the Run Time System (RTS), and the need for additional space reserved for dynamic memory allocation.

The use of dynamically allocated memory introduces the need for reclamation. For the purposes of this discussion, the terms storage allocation, storage deallocation, and storage reuse are considered to be within the domain of an Ada application. The terms memory allocation and (automatic) memory reclamation are considered to be in the RTS domain. Requirements for RTS dynamic memory allocation may occur implicitly and explicitly during execution of an Ada program. Allocation is implicit in support of procedure calls, task activations, etc. It is controllable only indirectly through careful structuring of program modularity and selective use of language features. Allocation is explicit through use of the "new" allocator, the use of which, of course, is optional. Whether implicitly or explicitly, RTS must manage

(allocate and reclaim) the system memory involved. However, exactly what the RTS does and how it does it may vary among products, product versions, and target machines.

Automatic memory reclamation of dynamically allocated objects can be a time-consuming asynchronous activity occurring at unpredictable intervals. This situation may not provide a good environment for a very time-critical real-time application. However, in an embedded real-time system having relatively little free memory space, the need to reuse deallocated storage is likely to be crucial to the long-term successful operation of the system. Without RTS storage reclamation or application reuse of storage for explicitly allocated objects, space associated with deallocated objects may not become available until expiration of the scope containing the definition of their access types. If the access type definitions occur in the outermost scope of the program, storage reclamation may never be done, and the application may eventually abort (raising the Storage_Error exception) due to memory space exhaustion.

It is important, therefore, that the design of an application provide an explicit memory management strategy to increase the predictability of memory usage. Running out of memory is a problem that can be very difficult to detect if it is a slowly developing condition. The longer it takes for the application to fill all its memory space, the less likely it is that formal testing will reveal the problem before the system's delivery and deployment.

Ada has no mechanism to ensure that deallocated storage is reclaimed. The explicit deallocation of the object designated by an access value can be achieved by calling a procedure obtained by instantiation of the predefined generic library procedure UNCHECKED_DEALLOCATION [RM 4.8-12]. Unfortunately, the Ada Reference Manual is inconsistent regarding the reclamation duties of an implementation, wavering between "may (but need not) reclaim" [RM 4.8-7] and "is to be reclaimed" [RM 13.10.1-5]. However, the current consensus seems to be that the definition of UNCHECKED_DEALLOCATION allows but does not require that the storage occupied by the designated object be reclaimed [RAT86 6.3.7 and AALC AI-00356]. It requires only that the object become inaccessible. Hence, simply assigning the null access value to the procedure argument is a legal implementation of UNCHECKED_DEALLOCATION. In the absence of knowledge concerning a specific Ada implementation, that is all one can count on.

The effect of a call to UNCHECKED_DEALLOCATION on storage reclamation, to whatever extent it is performed by the RTS, is unaffected by the existence of multiple (alias) access values referencing the object being deallocated. The deallocation is by definition unchecked, and the implementation is free to reclaim the storage immediately if not otherwise constrained. Therefore, the application must not assume that any non-null access value

will prevent storage reclamation for the designated object, and must be aware that use of alias access values may result in erroneous execution.

Access types can reference either data objects or task objects, the latter having different deallocation characteristics. When the argument in a call to procedure UNCHECKED_DEALLOCATION references a task object, the call has no effect on the task designated by the value of the argument. The same holds for any subcomponent of the task object, if that subcomponent is itself a task object (RM 13.10.1-8). Therefore, controlling reclamation of task object storage requires achieving control over the expiration of the containing program scope.

The guidelines below address approaches to the safe control of storage reclamation in real-time applications.

RTS Dependency Guideline 4.4-1: Use a storage management package that implements application-controlled storage reuse.

Discussion: Two major approaches to the design of such a storage management package derive from a decision either to allocate statically at one time all the storage that will ever be needed or to allocate dynamically and accumulate the storage on an as-needed basis. An example of a storage management package designed around the first approach is presented in [KOW87]. As presented, storage management support is provided in a generic library package, with the stored object type being a generic formal parameter of the package. A pool of storage is defined at the point of generic instantiation, and objects are allocated/deallocated using one of the Allocate or Free subprograms. The major techniques adopted are Mark/Release, to allow dynamic control over entire blocks of storage; Save/Free, to retain access to selected "saved" objects within the released block; and Reference Counting, to recycle unreachable storage by maintaining a count of "live" (possibly alias) references to an object with reuse initiated when the count drops to zero.

An example of the second approach is in [MEN87]. The author presents two variations: (1) an abstraction-specific model oriented toward a single object type and providing control over the creation of and reference to aliases and (2) an abstraction-independent model whose purpose is to export only dynamic memory management without controlling the object type or the aliasing problem. In either case, initial and additional storage is obtained through use of the "new" allocator. When the storage is no longer needed, it is set aside on a free list for reuse. Thus the amount of system memory used expands as the object (e.g., a linked list) grows, but does not decrease when the object shrinks. The difference remains allocated as far as the RTS is concerned, but is available on the free list for any future reexpansion of the object.

RTS Dependency Guideline 4.4-2: Use static allocation as much as possible to provide greater predictability of storage use and allow for better compiler optimization.

Discussion: Ada compilation systems commonly use heap storage for manipulation of unconstrained arrays. Using constrained arrays wherever possible reduces the chances of unexpected memory depletion or memory reclamation overhead. If constrained arrays adversely affect memory usage, the problem will probably surface during system development rather than becoming an emergency in the field. In addition, better optimization and the lack of RTS storage reclamation processing both contribute to faster system operation. The degree to which this strategy can succeed depends on the amount of memory available and the extent to which the application can be structured to allow safe reference to the objects involved.

RTS Dependency Guideline 4.4-3: Perform bit packing to save storage space, and apply storage limits on a per-access-type basis to gain firmer control over dynamic memory allocation.

Discussion: The bit size of an object can be specified by referencing its type name in conjunction with the SIZE attribute in a length clause. The space saving actually realized may be implementation-dependent. Further control over the use of memory by multiple occurrences of a specific object is also available. The total amount of storage available for a collection of objects of an access type (or for task activation) can be set by using the STORAGE_SIZE attribute in a length clause. If the application attempts to allocate more objects than will fit into the space reserved with the length clause, the STORAGE_ERROR exception will be raised. This result, analogous to raising the CONSTRAINT_ERROR exception for exceeding the index range of an array, may indicate that the application is not allocating the objects as intended.

RTS Dependency Guideline 4.4-4: Use the CONTROLLED pragma to eliminate the possibility that RTS-controlled storage reclamation will use system resources during execution of very time-critical sections of code.

Discussion: If a particular Ada implementation performs storage reclamation, there is a provision for delaying that activity. The pragma CONTROLLED informs the implementation that automatic storage reclamation must not be performed for objects designated by values of the specified access type, except upon leaving the innermost block statement, subprogram body, or task body that encloses the access type declaration, or after leaving the main program (RM 4.8(b)).

RTS Dependency Guideline 4.4-5: Consider structuring active tasks that have short lifetimes (relative to the lifetime of the main program) within scopes that can be forced to expire as necessary, thereby allowing the storage allocated to the task to be reclaimed.

Discussion: An example of this technique is described in [LEF87]. The basic concept is to declare a "dynamic" task within a recursive "master" block (block statement or subprogram), which is itself declared within and activated by a "base" task. A separate "access controller" task performs as the interface between the main application code and the "base" task by synchronizing task creation/deallocation commands from the application with recurse/unrecurse queries from the "master" block. Although this approach is not without its limitations (e.g., it requires time and space overhead to establish the additional base structures), it can apparently provide memory-efficient tasking in active real-time systems. Lefebvre claims that his implementation of this dynamic tasking utility "is currently being used within a large-scale real-time Ada project." Unfortunately, because the implementation of this utility is not trivial, it is not feasible to present a simple example here. The correct operation of the utility has not been verified.

REFERENCES

[AALC] Approved Ada Language Commentaries, Ada Letters, AI-00356/08, vol. IX, no. 3, Spring 1989.

[ACM77] Toward a Discipline of Real-Time Programming. Comm. ACM Volume 20 number 8, August 1977.

[ACM87a] Catalog of Interface Features and Options. ACM SIGAda Ada Runtime Environment Working Group, 1987.

[ACM87b] International Real-Time Ada Issues Workshop. ACM SIGAda Ada Letters Volume VII Number 6, 1987.

[ACM88a] International Real-Time Ada Issues Workshop. ACM SIGAda Ada Letters, Volume VIII Number 7, 1988.

[ACM88b] A Model Runtime System Interface for Ada. ACM SIGAda Ada Runtime Environment Working Group, August 1988.

[AUK88] The Virtual Node Approach to Designing Distributed Ada Programs. Ada Language UK Ada User, 1988.

[AVI84] Avizienis, A. and J.P.J. Kelly, "Fault Tolerance by Design Diversity: Concepts and Experiments," IEEE Computer, August, 1984.

[AVI85] Avizienis, A., "The N-Version Approach to Fault-Tolerant Software," IEEE Transactions on Software Engineering, December 1985.

[BAK85] Baker, T.P. and G.A. Riccardi, "Ada Tasking: From Semantics to Efficient Implementation," IEEE Software, March 1985.

[BAR84] Barnes, J.G.P., Programming in Ada, Addison-Wesley, 1984

[BRI73] Brinch Hansen, P., Operating System Principles, Prentice-Hall Inc., Englewood Cliffs, N.J., 1973.

[BUH84] Buhr, R.J.A., System Design with Ada, Prentice-Hall, 1984

[BUR87] Burger, T.M., and K.W. Neilsen, "An Assessment of the Overhead Associated with Tasking Facilities and Task Paradigms in Ada," Ada Letters, vol. VII, no. 1, January/February 1987.

[CEC84] Real-Time Systems In Ada, U.S. Army CECOM/CENTACS, 1984

[CEC88] Real-Time Technical Interchange Meeting: Real-Time & Reuse Working Group, US Army CECOM/CSE, July 1988.

[CEC89] Final Report - Real-Time Ada Demonstration Project, US Army CECOM/CSE, May 1989.

[CHE88] Cheng, J., K. Araki, and K. Ushijima, "Tasking Communication Deadlocks in Concurrent Ada Programs," Ada Letters, September/October 1988

[CMUSEI] Sha, L., and Goodenough, J., Real-Time Scheduling Theory and Ada, Technical Report CMU/SEI-89-TR-14 ESD-TR-89-22, Software Engineering Institute and Carnegie-Mellon University, April, 1989

[CSC86] Ada Reusability Study. Computer Sciences Corporation, Technical Report SP-IRD 9, August 1986.

[CSC87] Ada Reusability Handbook. Computer Sciences Corporation, Technical Report SP-IRD 11, April 1987.

[CUP88] Ada for Distributed Systems. Cambridge University Press, the Ada companion Series, 1988.

[HAB80] Habermann, A.N. and A.R. Nassi, Efficient Implementation of Ada Tasks, Department of Computer Science, Carnegie-Mellon University, 1980.

[HOA74] Hoare, C.A.R., "Monitors: An Operating System Structuring Concept," Communications of the ACM, vol. 17, no. 10, October 1974.

[KNI87] Knight, J.C., and J.A.A. Urquhart, "On the Implementation and Use of Ada on Fault-Tolerant Distributed Systems," IEEE Transactions on Software Engineering, May 1987.

[KOW87] Kownacki, R., and S.T. Taft, "Portable and Efficient Storage Management in Ada," Ada Letters, Using Ada: ACM SIGAda International Conference, December 1987.

[LEF87] Lefebvre, P.J., "Reclamation of Memory for Dynamic Ada Tasking," Ada Letters, Using Ada: ACM SIGAda International Conference, December 1987.

[LEV89] Levine, G., "Controlling Deadlock in Ada," Ada Letters, May/June 1989

[MEN87] Mendal, G.O., "Storage Reclamation Models for Ada Programs," Ada Letters, Using Ada: ACM SIGAda International Conference, December 1987.

[RAN75] Randell, B., "System Structure for Software Fault Tolerance," IEEE Transactions on Software Engineering, June 1975.

[RAT86] Rationale for the Design of the Ada Programming Language. Honeywell Systems and Research Center/ Alsys, Inc., 1986.

[RM] Military Standard: Ada Programming Language. Department of Defense, ANSI/MIL-STD-1815A, January 1983.

[SHU88] Shumate, K., Understanding Concurrency In Ada, McGraw-Hill, 1988

[WEN78] Wensley, J.H., et al., "SIFT: Design and Analysis of a Fault-Tolerant Computer for Aircraft Control," Proceedings of the IEEE, October 1978.